



**Programming
Techniques**

Simulation

Blaise W. Liffick, Editor

Volume 2

OTHER BYTE PUBLICATIONS

All **PAPERBYTE®** Books contain programs in machine readable object code in the **PAPERBYTE®** bar code format:

RA6800ML: An M6800 Relocatable Macro Assembler Jack E. Hemenway
LINK68: An M6800 Linking Loader Robert D. Grappel & Jack E. Hemenway
TRACER: A 6800 Debugging Program Robert D. Grappel & Jack E. Hemenway
MONDEB: An Advanced M6800 Monitor-Debugger Don Peters
SUPERWUMPUS Jack Emmerichs
Tiny Assembler 6800, Version 3.1 Jack Emmerichs

Other BYTE BOOKS,™ collections of favorite articles from past issues of **BYTE** magazine, plus new material and addenda:

Programming Techniques: Program Design Blaise W. Liffick (ed.)
Ciarcia's Circuit Cellar Steve Ciarcia
The BYTE Books of Computer Music Christopher P. Morgan (ed.)

**Programming
Techniques
Volume 2**

Simulation

Blaise W. Liffick, Editor



"BOOKS OF INTEREST TO COMPUTER PEOPLE"

A DIVISION OF BYTE PUBLICATIONS, INC.

70 Main St., • Peterborough, N.H. 03458 • (603) 924-7217

The authors of the programs provided with this book have carefully reviewed them to ensure their performance in accordance with the specifications described in the book. Neither the authors nor BYTE Publications Inc, however, make any warranties whatever concerning the programs, and assume no responsibility or liability of any kind for errors in the programs or for the consequences of any such errors. The programs are the sole property of the authors and have been registered with the United States Copyright Office.

Copyright © 1979 BYTE Publications Inc. All Rights Reserved. Portions of this book were previously Copyright © 1977 or 1978 by BYTE Publications Inc. BYTE and PAPERBYTE are Trademarks of BYTE Publications Inc. No part of this book may be translated or reproduced in any form without the prior written consent of BYTE Publications Inc.

Library of Congress Cataloging in Publication Data

Main entry under title:

Programming techniques.

CONTENTS: [1] Program design.—v. 2. Simulation.

1. Electronic digital computers—Programming.

2. Computer simulation. I. Liffick, Blaise W.

QA76.6.P7518 001.6'424 78-8649

ISBN 0-931718-12-0 (v.1)

ISBN 0-931718-13-9 (v.2)

Printed in the United States of America

TABLE OF CONTENTS

FROM THE EDITOR	v
ARTIFICIAL INTELLIGENCE	
What Is it?	
Richard Rosenbaum (BYTE Magazine April 1977)	3
An Evolutionary Idea, Part 1	
Michael Wimble, (BYTE Magazine May 1977)	9
An Evolutionary Idea, Part 2	
Michael Wimble (BYTE Magazine June 1977)	15
Natural Language Processing and Small Systems	
Harry Tennant (BYTE Magazine June 1978)	23
SIMULATION OF MOTION	
An Improved Lunar Lander Algorithm	
Stephen Smith (BYTE Magazine November 1977)	35
An Automobile Suspension	
Stephen Smith (BYTE Magazine December 1977)	41
Model Rockets and Other Flying Objects	
Stephen Smith (BYTE Magazine January 1978)	45
Extended Objects, Applications for Boating	
Stephen Smith (BYTE Magazine February 1978)	51
Microcomputer Flight Simulation	
F.R. Ruckdeschel	57
EXPERIMENTATION	
Robot Simulation of Microcomputers	
John Webster (BYTE Magazine April 1978)	81
Linear Circuit Analysis	
Leonard Anderson (BYTE Magazine October 1978)	87
Electronic Circuit Simulation	
Robert Arp	101
About the Authors	123
Acknowledgements	125

FROM THE EDITOR

Programming Techniques is a series of books designed specifically to help the personal computer enthusiast by making programming easier and more enjoyable. This is done by providing articles which detail successful techniques for designing and implementing programs. The first volume in this series, **Program Design** (ISBN 0-931718-12-0), provides a look at several different methods for designing programs more efficiently and effectively. Included in the topics covered are structured program design, modular programming techniques, program logic design, designing tables, and binary tree processing.

Volume 2 of **Programming Techniques** is **Simulation**. Its purpose is to familiarize the reader with both a general overview of the vast area of computer simulation as well as details of specific types of simulations. The term *simulation* can cover a lot of territory. But for this book I have chosen only three categories: **Artificial Intelligence (AI)**, **Motion**, and **Experimentation**.

I felt a brief introduction to AI was called for in this book, even though it is a complex subject of research unto itself, simply because of the large amount of research being done in the area. By no means do I suppose that this book contains any in-depth analysis of the world of AI. But it does provide a good introduction to that area for those unfamiliar with it. Some may question whether or not articles on AI belong in a book on simulations, but introducing the subject here could lead the reader to other areas such as game theory or robotics, which are indeed "true" simulations.

When most people think of computer simulations, they usually think of the modelling of physical activities — thus the section on **Simulation of Motion**. Several different types of motion are looked at, the main one being flight (for some reason a favorite among personal computer experimenters). These articles offer good models on which to base detailed simulations in many areas.

The final section on **Experimentation** is by far the most complex. It first gives an excellent example of a resourceful twist on the usual types of simulations: the simulation of a robot (a simulation of a simulation?). The rest of the section, however, is devoted to the analysis and simulation of electronic circuits. I personally find this the most exciting section because of what it represents: a return to the age of the home experimenter. It is no longer critically necessary to own vast amounts of exotic chemical or electrical apparatus. Many of the functions of experimental equipment can be simulated with even the simplest of microcomputer systems. So for the price of a home computer, you get not only a computer science laboratory, but also a chemistry lab, physics lab, electrical engineering lab, mechanical engineering lab, biology lab . . .

Blaise W. Liffick
Editor

ARTIFICIAL INTELLIGENCE

Artificial Intelligence:

What Is It?

Richard Rosenbaum

If you are a typical computer hobbyist, you have probably spent (or will spend) a fair amount of time playing games — computer games, that is. Perhaps you have written computer programs to play space war, backgammon, even chess. What techniques are used in programming computer games, particularly those requiring “intelligent” decisions by the machine? The answer to this question is in the realm of artificial intelligence.

Perhaps you are trying to program a microprocessor to recognize or interpret signals from a video camera, or would like to automatically translate Morse code to printed characters. Both of these problems concern artificial intelligence, specifically in the field of pattern recognition.

What is artificial intelligence? For that matter, what is intelligence? These questions can easily lead us into a philosophical debate, but that is not the purpose of this discussion. My interest is how artificial intelligence, frequently referred to as AI, can help us solve problems such as those just outlined. What these problems have in common is the substitution of a computer (artificial mechanism) for what are ordinarily human tasks (generally assumed to require intelligence). So for the sake of expediency, let's define artificial intelligence as the mimicking of human behavior and decision making by computer.

Today, work in artificial intelligence is divided into a number of areas. Two of them

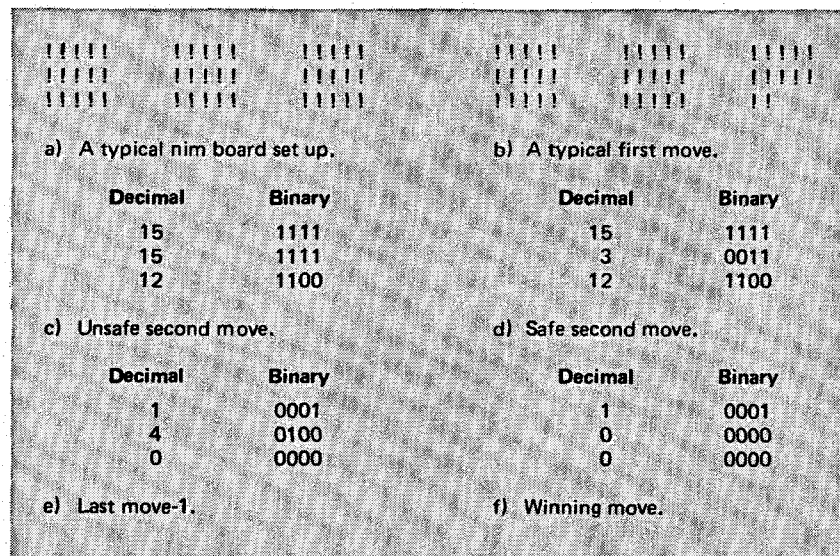


Figure 1: One version of the game of nim starts with three piles of 15 counters as shown in figure 1a. Players take turns removing any number of counters, at least one, from any one pile. The player who takes the last counter loses. In the case illustrated in figure 1b the first player took three from the last pile. Figures 1c and d illustrate some techniques used in developing an infallible play of nim. The first step is to represent the number of counters in each pile in binary. Then you must determine the number of 1s in each binary column. If the number is odd, as in figure 1c, then you are in an unsafe position. You should make your move so as to leave an even number of 1s in each column as shown in figure 1d. This is known as a safe position. A point is eventually reached, as in figure 1e, where there are only two piles left and only one of them has more than one counter. The winning strategy is to take all of the counters from the pile with more than one. This causes the other person to take the last counter, as shown in figure 1f. Using these methods, an unbeatable game of nim can be developed.

have been mentioned: game playing and pattern recognition. Others include generalized problem solving, natural language comprehension and translation, and robotics. These will be examined in order.

Game Playing Programs

It is probable that once the beginning computer hobbyist learns basic programming fundamentals, the first attempt at a big program will be an entertaining one. How many of us, once we have BASIC up and running, quickly type in our copy of Star Trek? We all have some interest in computer games.

While many games use artificial intelligence concepts in their programming, not all of them do. For instance, Star Trek usually consists of a program that keeps track of time, energy levels, scores and other parameters. The only decision to be made is generally, "Which way do I fire?", which only requires use of a simple formula. Of course, there are exceptions where some super program simulates the alien cunning of a Klingon commander.

Artificial intelligence techniques are required, however, for the automation of most traditional multiple player games. These range from the simple binary strategy of nim (see figure 1) to the complexities of chess. The case of nim illustrates one extreme of an intelligent program: the perfect player. The game is simple enough to develop a fast technique to determine the best move. Such is not the case in chess. The important point here is that both chess and nim are deterministic; ie: if you have enough time, you can go through every possibility that can arise. However, with games as complex as chess, a program would take a tremendous amount of time to check all the possible moves. This is where artificial intelligence techniques come to the rescue.

Let us examine how a human player makes decisions in a game such as chess. This technique, saying, "How would I do it?", is invaluable in artificial intelligence work. Obviously, we do not go through every possibility. What we do instead is to make trial moves and evaluate the board situation at that point. If the situation is too dangerous, we discard that possibility and check others until we make a decision.

So we need an evaluation function. The quality of this function will strongly influence how good a game the computer can play. In 1967, A Samuel wrote a program to play checkers. In it, he used an evaluation formula that took into account such parameters as number of pieces, number of kings, how far the kings were advanced, etc. Samuel, however, did not know how to weight the various items to determine their relative importance. So he made a set of trial values and incorporated into the program a mechanism that changed these values if it was losing, in order to try to find an optimum strategy. In this way, he wrote a program which "learned" to play better checkers!

Of course there is more to playing a game than just evaluating positions. We must generate trial moves in some orderly fashion. The most efficient way to do this is to make a tree of possible moves as shown in figure 2 for example. This figure is called a tree because of the way it resembles an upside down tree with branches.

As can be seen, trees can rapidly expand as they grow downward. The tree shown in figure 2 has 1.2×10^{56} branches. With the limited memory space of a typical micropro-

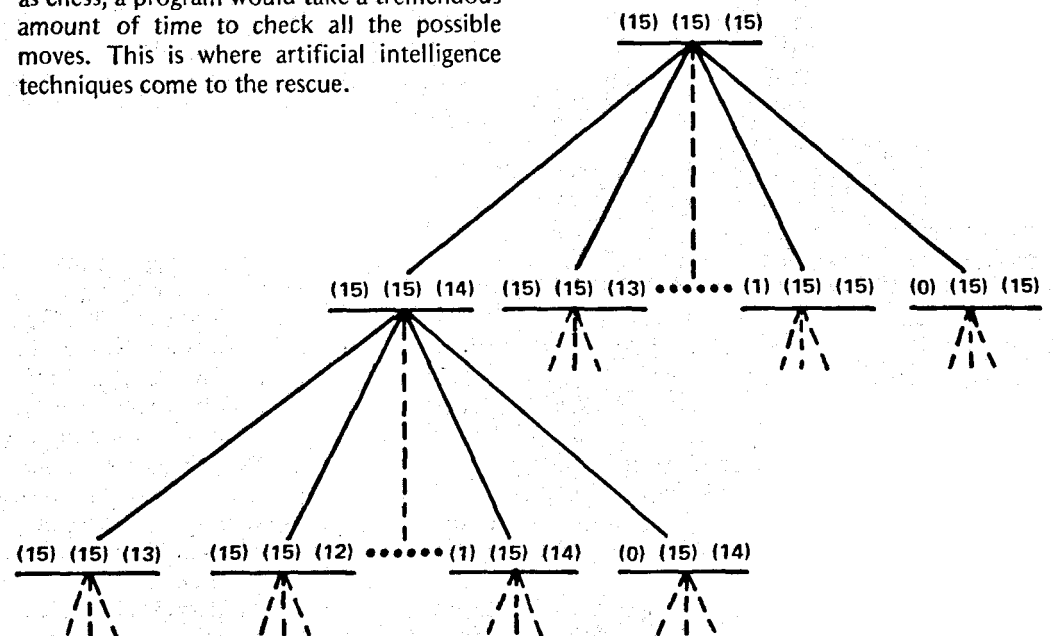
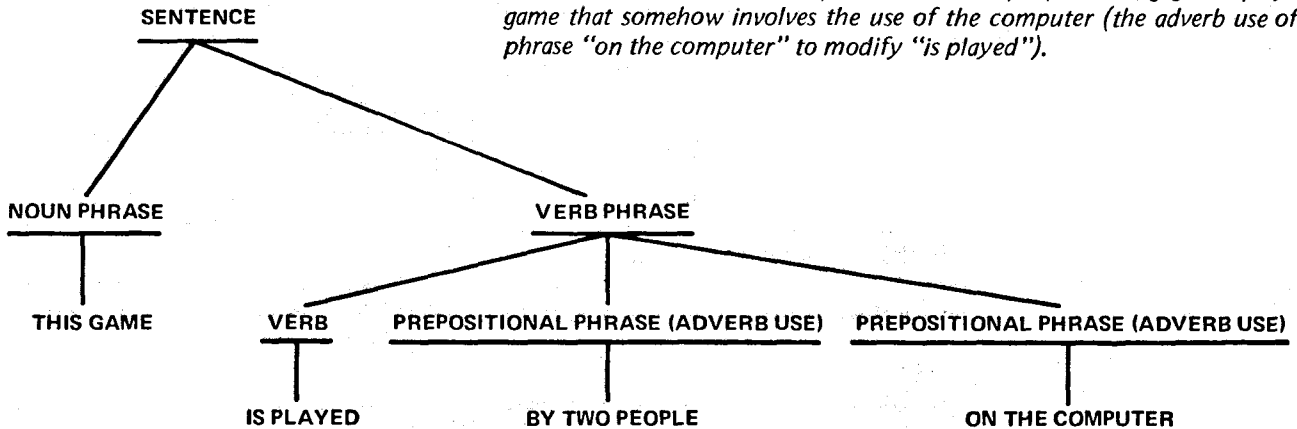


Figure 2: This is the beginning of a game tree for the game of nim as discussed in figure 1. Only a few of the branches are shown for the sake of clarity. Using this version of the game there are 45 different ways to play the game!

Figure 3: A syntactic analysis of the sentence, "This game is played by two people on the computer." This particular parse of the sentence, resolving the sentence into its components, means that two people are engaged in playing a game that somehow involves the use of the computer (the adverb use of the phrase "on the computer" to modify "is played").



cessor, we must find some way to minimize its size. One way is to copy the human player, and not expand any branches that do not look promising, ie: do not have a high evaluation. Other aspects of artificial intelligence concern optimization of the evaluation, speeding up tree generation, and minimizing other memory requirements. Often, one only stores a small local region of the tree at any one time, using rules to generate additional segments.

Pattern Recognition

The influence of computer aided pattern recognition is affecting us more each day. Our checks have funny digits printed on them, grocery items have strange stripes; you also might have run into forms that required you to write characters exactly as indicated. A portion of artificial intelligence concerns itself with these tasks.

Pattern recognition is generally a problem of classification. We have either a single item which we wish to describe as having certain characteristics, or we have a group of objects which we wish to break into subgroups. The principal problem is determining if a particular object is in a certain class of objects. Hence, the field extends beyond merely recognizing characters on a piece of paper. Potentially, artificial intelligence techniques could help classify medical problems based upon symptoms, the field of computer aided diagnosis long found in science fiction literature.

Unfortunately, much of the work in this field involves higher mathematics which are beyond the scope of this article. However, some elementary principles may help you conduct some investigations, perhaps in computer analysis of video signals. A fundamental technique is to break your recogni-

tion problem into the simplest subproblems possible. For instance, consider the problem of analyzing faces. One would have to generate parameters for subclassifications such as color of eyes, hair, etc. In this way, each face is assigned a set of basic values, a prerequisite of any attempt to make comparisons.

Problem Solving

Problem solving may appear to cover everything under the sun but within artificial intelligence it refers to a specific set of tasks. In artificial intelligence, a problem is said to exist when we are in an initial state, desire to reach a goal state, and have a limited set of operations that can be used to get there. Game playing is actually a subset of this, in that the initial state is the present position, and the goal state is a win. Another example is theorem proving, in which the initial state is given (remember your geometry?) and the operations are the various axioms.

There have been many attempts to develop general methods for these types of problems. In fact, an early one was called GPS, General Problem Solver. The authors of this program, A Newell, H Simon, and J Shaw, developed many of the techniques in this field. The simplest one was a brute force method, similar to our discussion of checking every possible move in a chess game. They called this the "British Museum algorithm" in reference to the fact that a monkey placed in front of a typewriter will eventually write all the books in the British Museum. So much for that idea!

Of all the work done, the most popular idea was embodied in a program called the Fortran Deductive System. This program would take the initial and goal states, and develop a set of differences. It then would

methodically apply the various operations in an attempt to reach the goal. The difference it had from the brute force approach was that it knew which operations affected which differences. By the way, any technique that "intelligently" reduces a problem in complexity so that computation time is reduced is called a heuristic.

Natural Language Processing

In the early days of modern computers, the early 50s, one of the great optimistic hopes was to have automatic machine translation of text. It was thought that all you would have to do is have enough memory to store a dictionary. Today there are not many knowledgeable people who would make such a boast.

The problem with writing a translating system is the immense complexity of natural language. English, for example, is full of potential ambiguities. Does "Time flies!" mean a comment about how time is going by quickly, or does it mean a command that we should start chasing flies with a stopwatch? One can think of hundreds of such sentences.

A related problem is natural language comprehension. Wouldn't it be nice if we could converse with our processors in English instead of BASIC or assembly language? Unfortunately, the same problems are faced in this task. One way to alleviate the problem is to limit our speaking vocabulary. Monitors often do this to an extreme, limiting us to specifics such as examine and deposit. The most successful approach for larger systems involves syntax directed

methods. This means that each sentence is broken down until its meaning is understood. This method has a "hopefully it will work" attitude about it. Figures 3 and 4 illustrate this technique, showing how a single sentence can be broken down into different meanings. These methods are also applied to the design of computer languages, their interpreters, and compilers.

Robotics

This subject is last for good reason. The design of robots is largely based upon the preceding topics. For vision, if it is capable of it, a robot would use pattern recognition techniques. For performing tasks, problem solving is involved. Robotics is an application of artificial intelligence to hardware of mechanical systems as a control element.

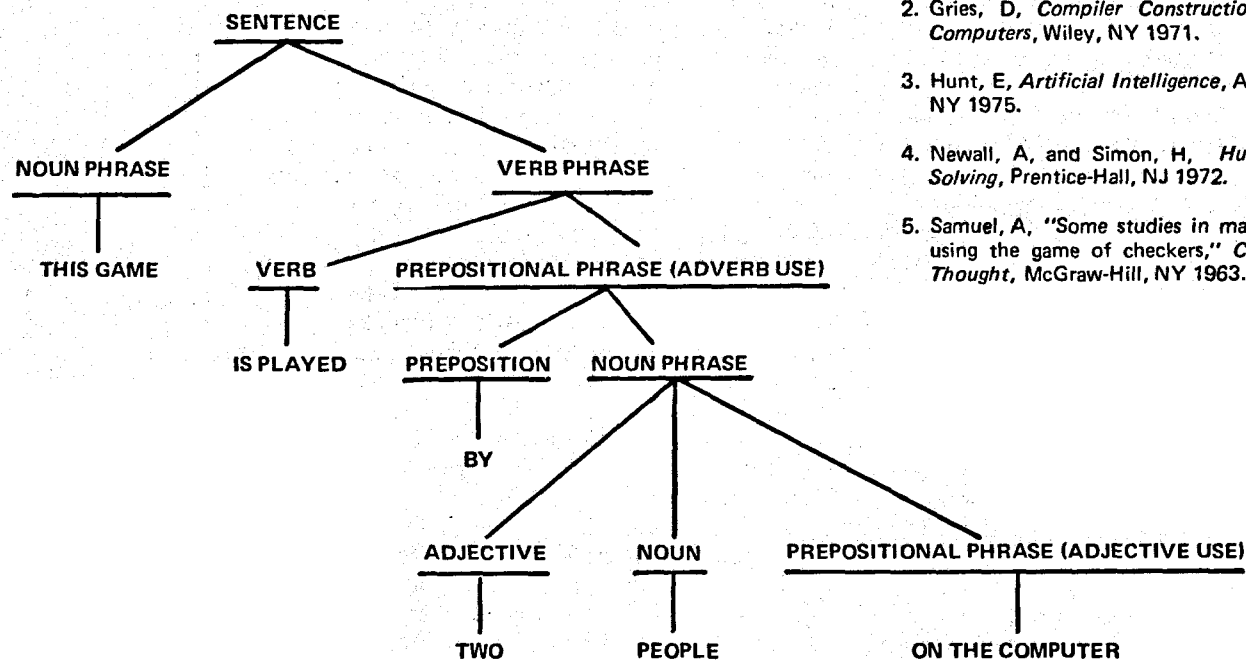
Conclusion

Artificial intelligence probably offers something for every programmer . . . certainly if you like to play games. But there are other aspects that are just briefly touched upon here. If we are building a large library of information, perhaps we would like to use an English-like inquiry system. The computer directed medical diagnosis could be extended to auto maintenance, or even computer repair. Whenever you want some of your thinking done for you, automate it with artificial intelligence.■

REFERENCES

1. Gelerntner, H, "Realization of a Geometry Proving Machine," *Computers and Thought*, McGraw-Hill, NY 1963.
2. Gries, D, *Compiler Construction for Digital Computers*, Wiley, NY 1971.
3. Hunt, E, *Artificial Intelligence*, Academic Press, NY 1975.
4. Newall, A, and Simon, H, *Human Problem Solving*, Prentice-Hall, NJ 1972.
5. Samuel, A, "Some studies in machine learning using the game of checkers," *Computers and Thought*, McGraw-Hill, NY 1963.

Figure 4: Another parse of the sentence of figure 3. This particular parse is evaluated as meaning that two people are on top of the computer (adjective use of "on the computer") and are engaged in playing a game. Both the parse of figure 3 and this one are valid and demonstrate a major difficulty that is encountered when trying to use computer evaluation of English or other natural languages.



This excerpt from a letter expressing a different point of view from the preceding article was received and printed in the January 1978 issue of BYTE Magazine.

Toward Smarter Machines

Many people seem to be missing the true meaning of artificial intelligence. Richard Rosenbaum declares that artificial intelligence is simply "the substitution of a computer (artificial mechanism) for what are ordinarily human tasks (generally assumed to require intelligence)." I must disagree.

The problem comes in defining intelligence.

In having the computer perform simple operations, such as the four operations of arithmetic, are we not substituting a computer for a human task? Is this artificial intelligence? It is rather simulated intelligence. The computer appears to be operating like a human mind. However, the computer employs a sequence of "mechanical" operations to arrive at a solution to the arithmetic problem. The human mind makes use of previously compiled knowledge, reorganizing it. The computer generates new data.

With consideration, one realizes that most human mental functions are based upon the manipulation of stored information. As data is received through the sensory systems, it is placed in the mind in the order perceived. As comparisons are made the information is copied into other areas of the brain. Associated images are stored together. Collections of data are called out again when the senses receive a piece of information common to the group. Mental structures

grow and are enhanced constantly; so grows the mind from the moment of birth to the instant of death.

Obviously, present day experimenters' computers do not have the capacity to handle information on the same scale as the human mind. However, all animal minds seem to function in essentially the same manner. As a matter of observation, emotional capacity and intelligence seem to have a direct relationship to the physical size of a particular species' brain. It may be possible to synthesize the mental functions of less intelligent animals; insects, rodents, primates. One might even consider synthesizing the infant human mind. While we may not actually create complete minds, it is possible to develop the basic foundations of self evolving intelligence.

Innovations in cybernetics are making it possible to give the computer necessary perceptual input. Vocal input and output devices and digital "eyes" are making it possible for the computer to hear, see, and stimulate its environment in order to obtain new information.

I would like very much to communicate with anyone interested in the study of artificial intelligence. Perhaps by coordination of efforts we can make more rapid progress.

Scott Jarol
3401 E Shaw Butte Dr
Phoenix AZ 85028

Artificial Intelligence, An Evolutionary Idea

Part 1: An Overview

Michael Wimble

Artificial intelligence is a subject that has fascinated people over the world for many years, but with the introduction of the computer this fascination is becoming a reality. I do not mean the intelligence depicted in so many science fiction books and sensational movies, but rather the slow and methodical application of computers to perform tasks formerly thought to require intelligence. To attempt to even briefly describe the areas and techniques of artificial intelligence research would certainly occupy several text books. The small computer user can, however, enter this exciting field and perform personal research and experiments by using the technique described in this article.

The technique is called "Artificial Intelligence Through Simulated Evolution" and its early use and implementation is described very well by Fogel, Owens and Walsh in their book of the same name, published by John Wiley and Sons, New York, 1966. I have chosen the technique for several reasons:

1. It can be simple to implement.
2. It is a powerful technique.
3. It is readily programmable on any small general purpose system.
4. It gives the user a quick and simple introduction to the realm of artificial intelligence.

The criterion just listed represent only the low end of application of this technique. Indeed some versions of programs using this technique perform very sophisticated functions using hundreds of hours of time on large computers . . . and even involve the use of artificial intelligence to analyze and optimize the very artificial intelligence process.

The Simulated Evolution Technique: What It Can Do

Before describing in detail the logic and theory of the simulated evolution technique, let us look at some early uses which are both interesting and will probably also be the initial uses of the personal computer. The technique is basically a goal directed pattern recognizer. That is, it performs some task whose success is dependent on the formation and recognition of patterns. The goal given to early programs was usually to predict some future event based upon past experience, although some other uses were in the areas of correlation analysis, image enhancement and feature extraction.

One example of an early use, which was more to debug the program than anything else, was to predict prime numbers. The computer was asked whether or not the next number was a prime. It then replied with a yes or no, and was told whether or not it

had predicted correctly. The computer then remembered this historical data and looked for patterns to be used in future predictions.

To show that this technique is not just some heuristic or statistical method, let's look at a slightly different but more interesting case, that of predicting earthquakes. In the prime number example, the goal was simply to predict whether or not the next number was prime. It was given a very simple "sense of values" that said making a good prediction was good and making a bad prediction was bad. By altering this sense of values, we change the goal.

For earthquake prediction, the goal is as follows:

- Predict when an earthquake will next occur.
- To make a good prediction is very good and to make a bad prediction is very bad.
- It is better, however, to predict a false alarm than to miss an actual earthquake.

Now let's compare the performance of the program when given the two different goals.

In the case of prime number prediction, after about the seventh observation the program realized that no multiple of two was prime. After about the seventeenth observation, the program realized that no multiple of three was prime. Unfortunately, as the numbers get larger the occurrence of primes becomes less frequent, at least for the first few hundred numbers. So the program rapidly settled into predicting that the next number will not be a prime, which is statistically very good.

In the case of earthquake prediction and using the same data as with the prime number case (ie: the occurrence of a prime number is the occurrence of an earthquake), the program acted in a much more sophisticated manner. In fact, a correct earthquake prediction was almost never missed, although 10% to 20% of its predictions were false alarms.

Finally, let's examine what is perhaps the most fascinating use of the technique: predicting people. One version of the program was set up as a game. Its goal was to predict which thing the opponent was going to do next. The opponent, a human player, was allowed to do one of two things, such as say true or false. The game was played in real time and was thus competitive. The goal of the opponent was to be unpredictable. Each time the computer correctly predicted the opponent I scored a win for the computer; otherwise it was a win for the opponent.

The opponent was free to choose any strategy as long as he was unaided. That is,

he could not use random number generators or tables or other such aids. He might, for instance, choose to be as random as possible right from the start or he might try to use a particular pattern until he felt the computer had discovered it and then to use a different pattern.

One would think that if a person could make one of two choices, and he/she made choices purely randomly, then the best one could correctly predict him/her would be 50% of the time. This is true. In practice, however, an unaided human cannot perform random actions; there is always some pattern or at least a bias. Moreover, it seems that a person in a competitive situation tries to find the pattern of his/her opponent and to then develop an antipattern pattern. One proof of this, in fact, is the performance of the computer. When played against several different players and requiring at least a hundred predictions before the opponent could end the game, the computer successfully predicted the opponent 55% to 85% of the time, depending upon the opponent. Typically, 65% of its predictions were correct.

The Simulated Evolution Technique: An Overview

I will now explain the general functioning of the program, first by analogy and then by actual detail. The analogy may be shocking to some, but bear in mind that it is *only* an analogy. I use it because, although it is technically inaccurate, it helps make the process clear and it provides some layman type concepts and explanations for the actual mathematical processes that occur.

As the name of the technique implies, the process involves creating an "organism" which then undergoes evolution. The organism has a survival instinct, and to survive it must successfully and continually achieve a goal. If at any time it fails, it is made to produce mutated offspring, each slightly different than the parent. These offspring along with the parent are then compared to determine which one is best able to achieve the goal, usually based upon historical data. The one best organism is then kept as a new parent and all the rest are destroyed. The new parent now survives until the next time it fails to achieve the goal.

What we then have is the simulation of thousands of years of evolution in a matter of minutes on the computer. Each organism has several basic facilities to aid in achieving its goal. It has an awareness of the goal via a matrix of values which serves both as the evolutionary selection criterion and as a sense of values or judgments. It is able to interact with the environment as in the case

of predicting the environment. Finally, it has a memory. Some programs allow unlimited memory while others may somehow restrict what can be remembered. Some programs even allow the organism to choose by itself what it wishes to remember.

The importance of memory is well illustrated by Fogel and his co-authors (page 1):

The distinction between knowledge and intelligence became clear; knowledge being the useful information stored within the individual, and intelligence being the ability of the individual to utilize this stored information in some worthwhile (goal directed) manner.

These characteristics: goal direction, environmental interaction and memory, when coupled with one other facet, are responsible for the program's "intelligent" behavior. The single most important characteristic of the program is that it is self-organizing. It starts out with no preconceived notions. The programmer's own experiences are not reflected in the programmed organism. The program is not limited to performing only a few particular goals. What does occur is that the program is given a goal and the organism alters itself, changing its own makeup in accordance with its own judgment and experience, always becoming better able to achieve the goal. The goal can be any of a wide variety of goals, prediction being only one of the simpler goals. The goal can even change while the program is running and the organism will alter itself to meet this new goal. With this in mind, consider Fogel's offering of a definition of intelligence (page 2):

More carefully stated, intelligence can be viewed as the ability of any decision making entity to achieve a degree of success in seeking a wide variety of goals under a wide variety of environments.

There are few restrictions on the program other than those explicitly or implicitly given by the programmer. The restrictions within which the programmer must operate are as follows:

1. The amount of computer memory available determines the maximum possible sophistication of the program.
2. The programmer will usually place limitations on the evolution process so that it will stop after some time period. Thus it will not take hours or days to make the next response.
3. The organism can recognize only a fixed number of input observation types. For the game described in part

2 of this article, the number is two. The amount of computer memory required is related to the square of this number and so is seldom greater than 16.

4. The organism can make only a fixed number of output response types. For the game to be described the number is two and is the same as the input observation types (ie: the output is a prediction of the next input).
5. The program operates sequentially. It makes a response, receives feedback which it uses to judge the "goodness" of its previous response, and reorganizes itself to make the next response.

The Simulated Evolution Technique: Detailed Representation

The actual program uses what is called a state space mathematical model. The model consists of a set of five elements thus:

$$\{S\}, \{I\}, \{O\}, \{T\}, f$$

where:

$\{S\}$ is a set of states

$\{I\}$ is a set of input symbols (input alphabet)

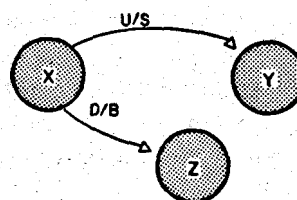
$\{O\}$ is a set of output symbols (output alphabet)

$\{T\}$ is a set of transitions

f is the current state

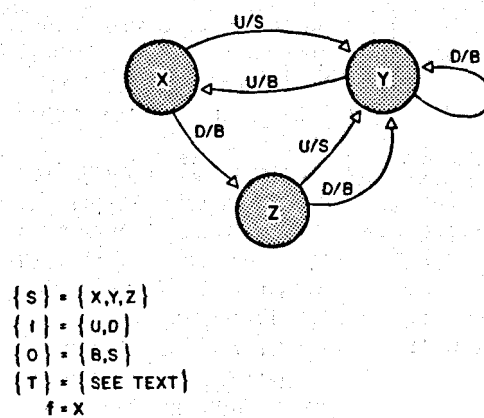
The brackets " $\{\}$ " denote sets, a notion often found in mathematical texts.

As an example, figure 1 represents a hypothetical model for a stock investor. The current state, f , is state X. The model says: The model is at state X and if the market price of the stock goes up (symbol U) then the investor will sell a block of the stock (symbol S) and go to state Y; otherwise if the market goes down (symbol D) then the investor will buy a block of the stock (symbol B) and go to state Z. This is represented in figure 1 as:

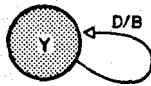


Assume that the market had gone up in the previous example, the current state

Figure 1: Hypothetical state space model of a stock investor. The letter codes U and P stand for the up and down fluctuations of the stock market prices. The letters B and S stand for the buying and selling of stock. The set T is the set of all the transitions of the model.



would now be Y and if the market price of the stock were to now go down the model says that the investor would buy a block of stock and remain at state Y. In figure 1 this is represented by:



For our pictorial representation of a state space model we have a set of states which is represented by the circles. The set of transitions is represented by the arrows. Alongside of each transition arrow is a pair of symbols separated by a slash. The symbol to the left of the slash is a symbol from the input alphabet and says that if the model is currently at the state connected to the tail of the arrow and this input symbol occurs, then this transition is to be taken. The symbol to the right of the slash is a symbol from the output alphabet and says that if this transition is taken, as per the previous statement, then this symbol is to be output and the new current state is the state connected to the head of the arrow.

State Space and Evolution

Given a state space model, we now look at the evolution process. There are five basic ways we can change the model; we can:

1. Add a new state (change the set $\{S\}$).
2. Remove an existing state (change the set $\{S\}$).
3. Change (repoint) a transition (change the set $\{T\}$).
4. Change the current state (change f).
5. Change an output symbol for a transition.

Changing an input symbol is not allowed. Every state must have exactly as many transitions projecting from it as there are possible input symbols. Thus when at a

particular state there is a valid response for every valid input.

Note that the set of input and output symbols cannot be changed during the run of a program. Technically there are ways of allowing this change but the programming becomes highly complex, so we assume that all possible observance types and all possible desired response types are known when the program begins. [In programming, as in physics or math, this is what is called a "simplifying assumption."]

Also, change type five is really not something that needs to be done randomly. There is a technique called deterministic optimization that assures that the output responses are the best possible for a given model. There is, however, a desirable change which can replace change type five and will be explained at the end of the next section.

The State Space Model as an Intelligent Organism

There are some interesting properties of the state space models that get developed via this simulated evolution. The first is the tolerance to noise. Also, the model is able to repair damage to itself. Finally, if the model is knocked out of synchronism with the environment it quickly locks back into synchronism.

When the model is given the goal to find a pattern, it may be constrained by the programmer or may have decided on the basis of previous inputs that the pattern for which it is searching is a simple pattern. The environment may then contain high levels of random signals (noise) but the model will automatically ignore these random elements, deciding that their pattern is too complex. It will then lock onto the underlying patterned signal and thus remove it cleanly from the noise.

If the model is suddenly damaged, such as by loss of power in a memory module, or even by programmed destruction of a state or a transition, the model tends to "heal" itself as new data is collected. It just so happens that the model is always striving to be a perfect representation of the environment. If this representation is changed, equivalent, say, to a person losing a leg, then the evolution process will tend to restore the damage to regain that optimum representation.

A common problem that does exist from this method of artificial intelligence is the loss of synchronism with the environment. What may be an otherwise perfect model may consistently yield incorrect predictions simply because it started out at the wrong state. Fortunately, as with the other problem types, the model has a tendency to lock

Figure 2: State space model in matrix representation. The model can be described by an n by m matrix. The number of states in the model is n and the number of input types is m . Each element describes the output response and the new current state to which the next move is made.

Current State	Input Observation			
	A	B	C	...
R	output: response A	output: response B	output: response C	
	new current state: R	new current state: R	new current state: R	
S	output: response A	output: response B	output: response C	
	new current state: S	new current state: S	new current state: S	
T	output: response A	output: response B	output: response C	
	new current state: T	new current state: T	new current state: T	
⋮				

current state = S

itself back in synchronism with the environment. Since, however, this problem can occur for so many reasons, we opt to provide an evolutionary function to speed up this lock. We then replace the old change type five with a new type, "advance the current state ahead by one state." We are now complete in the definition of the mutation types desired in our evolution simulation.

The State Space Model and Its Computer Representation

We now have sufficient knowledge to begin examining the computer implementation of this artificial intelligence by simulated evolution. Although the discussion will tend to be general, the emphasis is on using the technique as a two symbol pattern recognition game. This game would then allow the computer programmer to experiment on his/her own system with predicting primes, earthquakes, people or any of the virtually limitless areas of pattern recognition and feature extraction. Those programmers with a more extensive background in mathematics and automata theory and with larger computer facilities at their disposal may want to attempt more sophisticated programs with many symbol pattern recognition, advanced evolutionary techniques, or even using the intelligence of the program to improve upon the intelligence process.

The most important design decision for the program is the representation method of the model. Figure 2 shows that the model can be represented as an n by m matrix where n is the number of states in the model and m is the number of input types. Each element on the matrix consists of two pieces

of information, a symbol to output and the next current state. Thus, if the model is currently at state S and input symbol type B occurs, then row S and column B of the matrix defines the output response and the next current state.

At least five other pieces of information must also be maintained for the model, namely the first state, second state, the current state, value and the number of states currently in the model. The current state is used to derive the model's next response. The first state, second state and value are used by the evolution process when the model must be driven by historical data, as when determining which of the offspring during mutation is the best. The number of states currently in the machine is also used by the mutation or evolution routines, as will be seen later.

Table 1 details a workable representation of a two symbol pattern recognition model for a small computer. Each element in the n by m matrix consists of two bytes. One byte for each possible transition. For both bytes, the high order bit determines the output

Byte	Use	Bit	Use
0,1	State 1	0	Output response type: on is response type 1 off is response type 0
2,3	State 2		
$(n-1) * 2,$ $(n-1) * 2 + 1$	State n	1-7	New current state.

Table 1: The representation that is used in the computer model for the 2 symbol pattern recognition. Table 1a is the memory allocation for the matrix. Each state has two bytes with which to work. Each byte is broken into two sections as shown in table 1b. Bit 1 is used to signify the output response type, either response 0 or 1. The remaining seven bits are used to indicate the new current state.

symbol. For example, high order bit on could mean "next number is prime," or "an earthquake will occur next," or "the opponent will next say true." High order bit off would of course mean the opposite. The other seven bits in the byte contain the relative number of the next current state, state numbers ranging from 0 to 127. Decoding the model to make a prediction is easily done on most microprocessors and a typical sequence of instructions would be:

1. Load current state number into accumulator.
2. Shift left one place (equivalent to multiplying by 2).
3. Add input symbol type (a 0 or a 1).
4. The result of step 3 is the relative number from the beginning of the model of the byte representing the

transition to be taken. Load this relative byte into the accumulator.

- 5a. If high order bit is on, then the response (prediction) is type 1.
- 5b. If high order bit is off, then the response is type 0.
6. "AND" accumulator with hexadecimal 7F; the result is the new current state.

Part 1 of this article has described the general overall workings of the simulated evolution technique. Using the knowledge thus far presented, the reader is prepared to write his/her own artificial intelligence program for predicting earthquakes, primes or even people. In Part 2 this discussion will continue with a description of the implementation of a predictive game with which the readers can experiment.■

Artificial Intelligence, An Evolutionary Idea

Part 2: Implementation

Michael Wimble

As described in Part 1, there are five types of mutations that can be performed with the simulated evolution technique. A separate subroutine will be used for each mutation type. Four of the subroutines rely heavily on a subroutine which generates a random number between limits. (For those systems not already possessing random number generators, a box accompanying this article gives an algorithm to produce pseudorandom numbers by the power residue method.)

Figures 1 through 9 are flowcharts of the basic modules which were extracted from a fairly sophisticated system of FORTRAN programs. These are intended to serve as a starting point for the reader in implementing his/her own program. If there is sufficient reader interest, I would be happy to program and publish program listings of implementations for one or more small system processors, in any popular computer language.

The rest of this article then is a description of the modules used to implement a 2 symbol gaming program using the artificial intelligence technique. If more detail is needed I suggest you look first into the book *Artificial Intelligence Through Simulated Evolution* by L J Fogel, A J Owens and M J Walsh (John Wiley and Sons, New York, 1966) or send questions and a self-addressed stamped envelope to me. Although there are many flowcharts and the technical jargon may seem complex at times, I wish to emphasize that the programming is simple and the technique

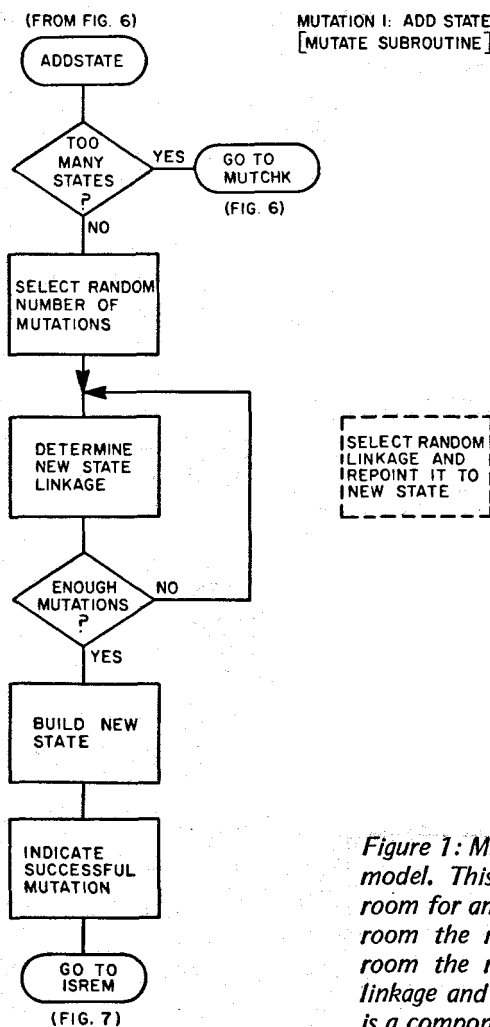


Figure 1: Mutation type 1, add a state to the model. This routine determines if there is room for another state. If there isn't enough room the model fails. If there is enough room the routine determines a new state linkage and then builds the new state. This is a component of the MUTATE subroutine.

can be implemented on any personal computing system with sufficient memory.

Mutation 1 – Add a State

Figure 1 describes the first mutation type. As shown, the number of states is first compared to some maximum number. Most programs will have a fixed amount of memory allocated for containing the model, and so a check is made to see if any memory is yet available for expanding the model. The

2 symbol version previously discussed can conveniently be implemented using two bytes per state with the maximum number of states being 127. The programmer must determine the internal representation of the model and the amount of memory available and then set a variable to represent the maximum number of states the model can hold.

The random number subroutine is called to provide an iteration counter. This counter is the number of randomly selected transitions that will be pointed at the new state. If no transitions were pointed to the new state then it would be an impossible state, ie: it could never be reached and could contribute no value to the model. Next, random transitions are changed to point to the new state which is to be created. Then, the new state is created. Finally ISREM is entered to remove any impossible states from the model.

Mutation 2 – Delete a State

Figure 2 describes the second mutation type. As with mutation type 1, a check is made of the number of states currently in the model. If there is only one state in the model, the mutation fails, since to delete the state would totally eradicate the model. If there is more than one state then the mutation will be able to proceed successfully.

Next, one of the states in the machine is randomly selected to be deleted. The only state that cannot be deleted with this subroutine is the state designated as the first state. To delete this state would result in the same mutation as mutation type 4 and is thus prohibited to this subroutine.

The actual deletion is accomplished in a rather roundabout manner. Every transition in the model is checked to see if it refers to the state being deleted. If so, it is repointed to some other random state number. The result of course is the creation of an impossible state, one that cannot be reached and is thus useless to the model. Entering the ISREM routine, as explained in the description of mutation type 1, will remove any impossible states from the model and so effect the actual deletion of the desired state.

Mutation 3 – Change a Transition

Figure 3 describes the third mutation type. The model is first checked for having more than one state. If there is only one state, then all transitions must necessarily point to that state, and so no change can be made and the mutation fails. Otherwise a random element in the model is chosen and its transition pointer is changed to point to

Figure 2: Mutation type 2, delete a state from the model. This routine checks to make sure that there is more than one state in the model or else it fails. If there is more than one state it deletes all references to a randomly chosen state thereby severing it from the rest of the model. This is a component of the MUTATE subroutine.

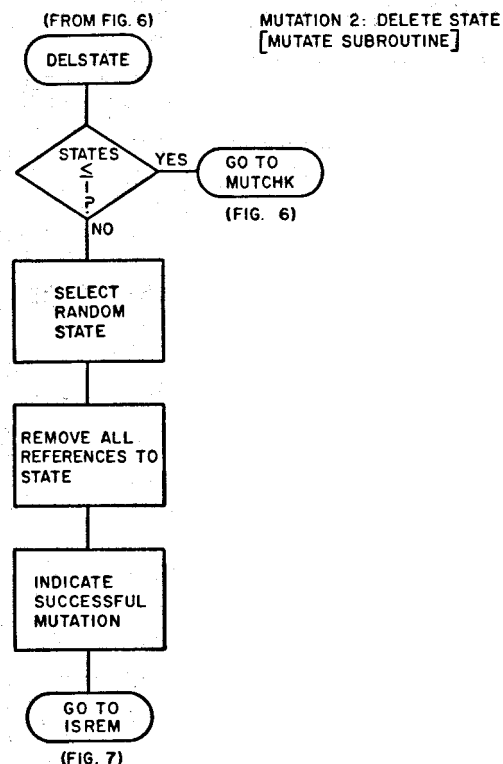
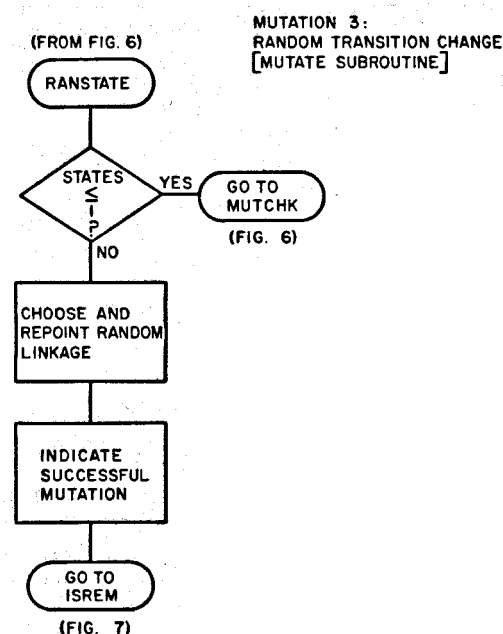


Figure 3: Mutation type 3, change a transition. This mutation will fail if there is only one state in the model. If there is more than one state a transition will be randomly chosen and repointed to another state from that to which it is presently pointing. This is a component of the MUTATE subroutine.



some other state. ISREM must be entered at the end of the routine since the changing of a transition may have resulted in the creation of an impossible state.

Mutation 4 – Change the First State Number

Figure 4 describes the fourth mutation type. Again, if there is only one state in the model, then it must necessarily also be the first state, and the mutation would then fail. Otherwise the first state number is set to any other state in the model. Also it is possible that an impossible state was created as the result of this mutation, so ISREM is entered to remove such impossible states.

Note that earlier discussion described the fourth mutation type as changing the current state number of the machine. The model must be driven by history from the first state when performing evolution, and it is seldom possible to work backwards from the current state number to determine the first state number. It is practical, therefore, to modify the mutation from change current state number to change first state number. The result of changing the first state number will usually end up changing the current state number.

Mutation 5 – Advance Model by One State

Figure 5 describes the simple but powerful mutation type 5. If the first state number is equal to the second state number of the model, then the mutation fails. Otherwise the first state number is set to the second state number.

Earlier discussion of mutation 5 described it as advancing the current state number by one. As with the case of mutation type 4, it is impractical to change the current state number directly, so the first state is advanced by one and evolution routines described later will run the model over its historical recall to result in the current state number actually being advanced by one state.

Mutation Control

Figure 6 shows the logic involved in controlling the mutation process. A random number is generated and used to select one of the five mutation types to be performed. Upon return from the mutation routine, a check is made for successful mutation. If the attempt was unsuccessful, a random selection is again made until a successful mutation is finally made. Finally, the routine OPTIMIZE is used to perform deterministic optimization and to assign a value to the model.

A Pseudorandom Number Algorithm

1. Test X, a 16 bit variable. If X is equal to zero, then set X to any number such as the time of day, or some other indeterminate number, and repeat step 1. This defines the initial seed of a pseudorandom sequence.
2. Multiply X by 259 to yield new value for X. Keep only the 16 least significant bits of the result. Increment X by 1.
3. If X is zero go to step 1.
4. Divide X by the input argument but do not destroy original value of X. The remainder of this result is a pseudorandom number between zero and one less than the input argument.

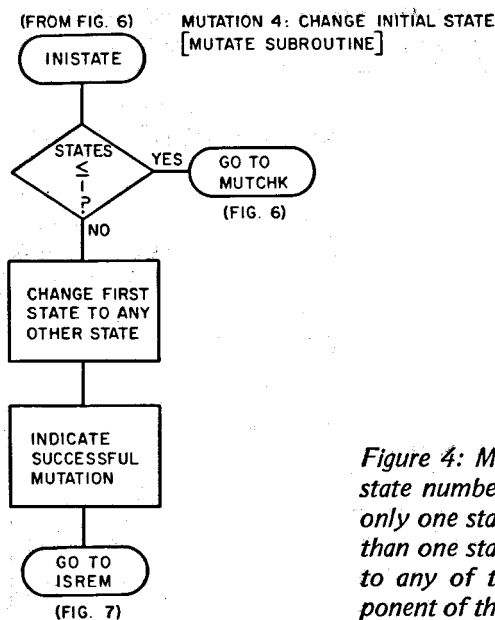


Figure 4: Mutation type 4, change the first state number. This mutation fails if there is only one state in the model. If there is more than one state the first state will be changed to any of the other states. This is a component of the MUTATE subroutine.

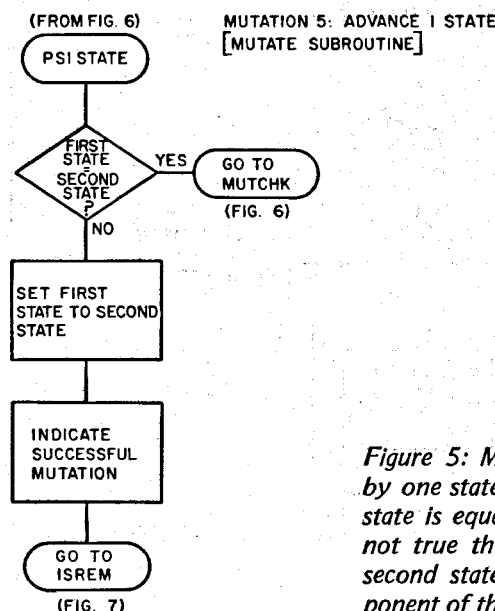


Figure 5: Mutation type 5, advance model by one state. This mutation fails if the first state is equal to the second state. If this is not true the first state is set equal to the second state of the model. This is a component of the MUTATE subroutine.

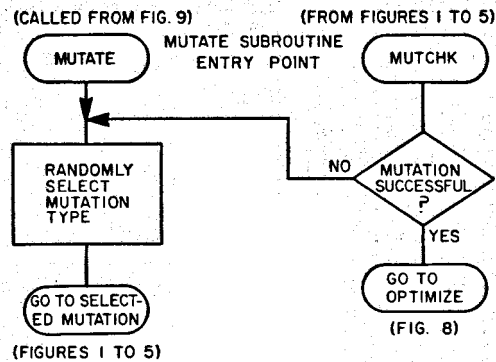


Figure 6: The *MUTATE* subroutine entry point. *MUTATE* chooses the type of mutation that will be performed using a pseudo-random number generator. At *MUTCHK* we check to see if the mutation was a success. If it was not then the *MUTATE* routine continues by choosing another mutation. (If at first you don't succeed . . .) If the mutation was a success then the optimization process cleans up the mutation by transforming it to its most usable form before returning control to the main program logic of *PREDMUT*.

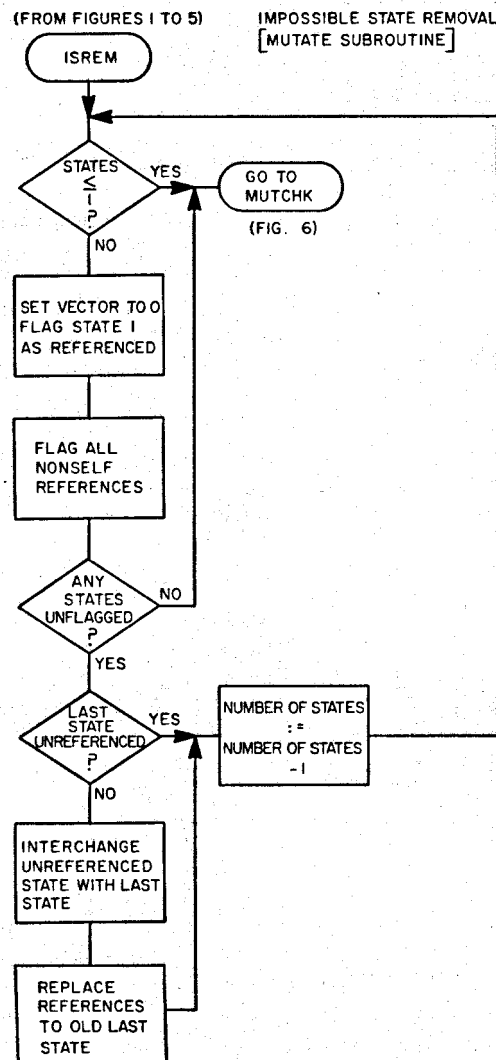
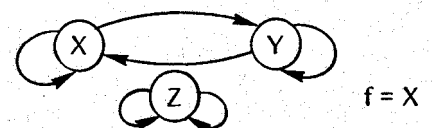


Figure 7: Routine *ISREM* removes any impossible states from the model. It performs this by checking all of the transitions to see that all of the states are referenced at least once. If it finds a state that is not referenced it will interchange that state with the last state of the model and delete the last position as valid. This is a house-keeping component of the *MUTATE* subroutine.

For those who implement this artificial intelligence game and would like to improve the intelligence forming process, the mutation control routine of figure 6 offers the greatest potential for improvement. The mutation selection process shown is simple and effective, but is essentially a brute force method. One of the first improvements I made to the program was to allow dynamic altering of the probability of selecting a mutation type. That is, if *ADDSTATE* was called successfully and resulted in a model with a higher value than before, then the probability of selecting *ADDSTATE* in the future was increased slightly. Similarly, if any mutation type was performed successfully and decreased the value of the model, then the probability of selecting that mutation type again was made slightly less. The most sophisticated versions of this artificial intelligence program use the very artificial intelligence process to optimize the selection of mutation types, thus aiding the evolution process.

ISREM – Remove Impossible States

Figure 7 describes the subroutine that removes impossible states. An impossible state is any state that cannot be reached. For example, the model:



has the impossible state Z. An impossible state cannot contribute to the value of a model but can hinder the model. If a model already had the maximum number of states permitted, but five of these states were impossible states, then the *ADDSTATE* mutation could not be performed until some states were deleted to make room.

The flowchart of figure 7 will not remove all impossible states. For instance, if in the figure above, the current state was changed from X to Z, then X and Y would now be impossible states instead of Z. The subroutine described will only find those states which are impossible solely because they cannot be reached from another state of the current model. In practice, this has never yet failed to eventually find all impossible states.

Looking at the flowchart then, the process is simple. First a table is set to zero. Every transition is then examined. If the head of the arrow points to a different state than the tail, then the entry in the table corresponding to the state pointed to by the head of the arrow is flagged.

After looking at all transitions, the table

prediction	actual occurrence			
	symbol ₁	symbol ₂	symbol ₃	...
symbol ₁	value ₁₁	value ₁₂	value ₁₃	...
symbol ₂	value ₂₁	value ₂₂	value ₂₃	...
symbol ₃	value ₃₁	value ₃₂	value ₃₃	...
⋮	⋮	⋮	⋮	⋮

Table 1: The matrix used to evaluate the optimized form of the new mutation. Each predicted input symbol is matched against the actual input. The value for that combination is then added to the total value of the model. In this manner, using historical methods, the new model can be compared to the older model to see if it is more efficient.

is examined. If any state in the model remains unflagged then it is an impossible state. If the impossible state is the last state in the model, then one need only decrement the variable denoting the number of states in the model to result in the deletion of the state. Otherwise the impossible state is first swapped with the last state in the model before the variable is decremented.

OPTIMIZE – Evaluate New Model

Figure 8 describes the most important segment of the program. OPTIMIZE performs the deterministic optimization mentioned previously, and evaluates the model with respect to the goal. Optimization and evaluation provide the criteria for the evolution selection procedure.

OPTIMIZE is conceptually simple, although often large of implementation. The model is made to perform with historical data. As each input observation is fetched and the appropriate transition made, the required output is noted. Thus for each transition we have a table of the number of times each output should have occurred for perfect prediction. The output symbol that should have occurred most frequently for a transition is then made the output for that transition. If the transition was never used during this historically driven run, then the transition is made to output the most frequently occurring output symbol over all of history.

After optimization, the model is rerun using the same historical data. Now, however, the optimized model is evaluated in terms of its ability to achieve the goal provided. The goal is expressed in a matrix of the form shown in table 1.

For each transition the prediction and

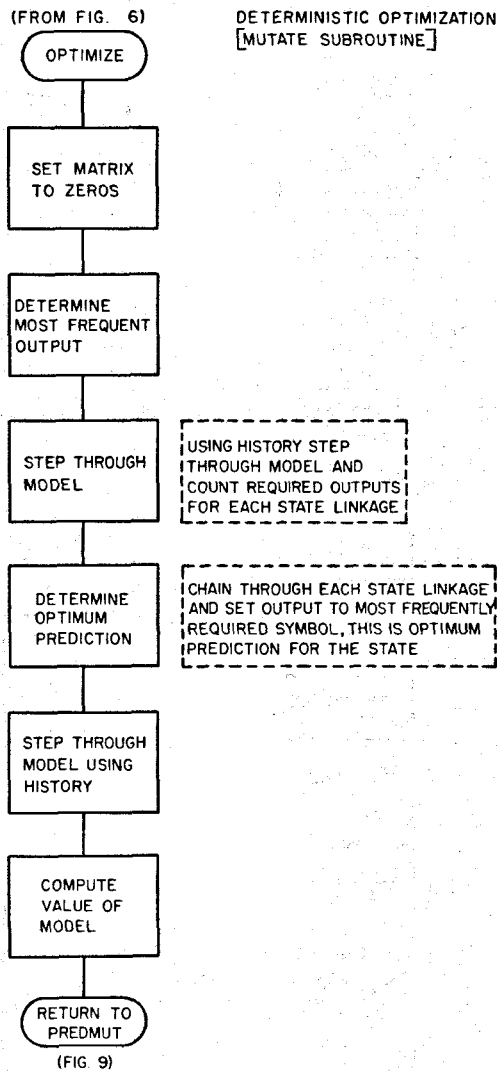


Figure 8: Routine OPTIMIZE determines the optimum prediction for each state of the new model. It will then evaluate the model using past history as a model to determine the value of the mutation. This is the final processing of the MUTATE subroutine of the program.

actual occurrence are used to extract a value from the table. The sum of these values is the value of the model.

For the purpose of predicting primes, the matrix is defined in table 2a, while for earthquakes the matrix is defined as in table 2b. You see then that each accurate prediction of an actual earthquake or no earthquake adds 1000 to the value for the model. To predict an earthquake when one does not occur adds 500 to the value of the model, while missing an earthquake prediction adds nothing to the value of the model.

PREDMUT – Putting It All Together

Figure 9 shows the mainline logic for the program. The flowchart should be obvious

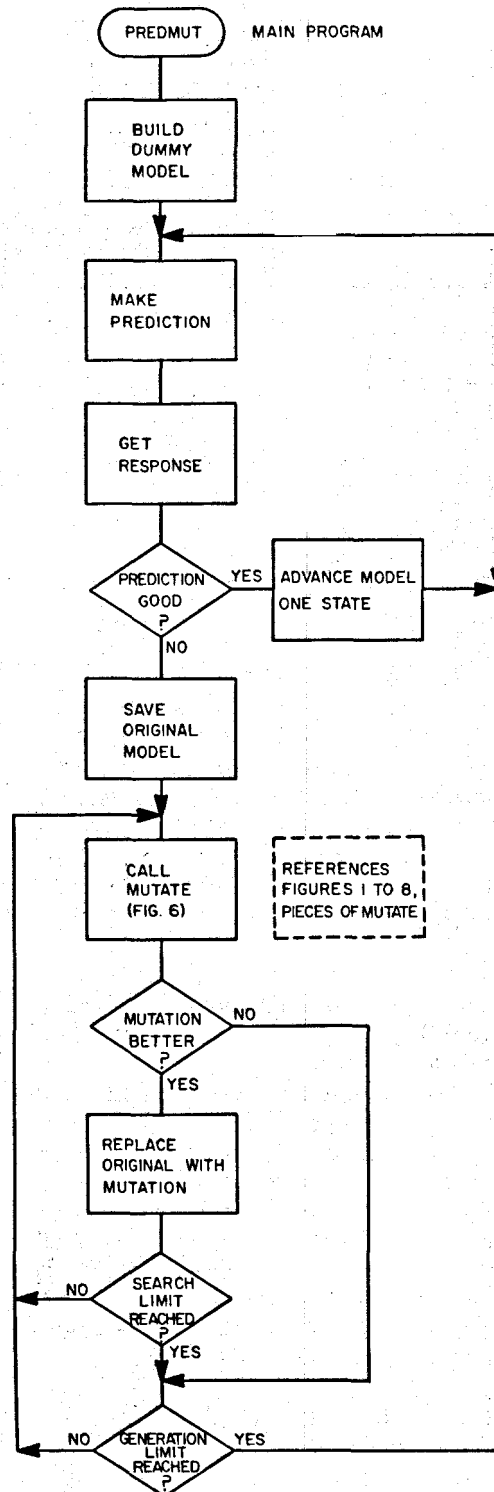
(a)

prediction	actual	
	prime	not prime
prime	1	0
not prime	0	1

(b)

prediction	actual	
	earthquake	no earthquake
earthquake	1000	500
no earthquake	0	1000

Table 2: Two tables which illustrate examples of goals used to evaluate the models using historical data methods. Table 2a shows that when a correct guess of either prime or not prime is made a value of 1 is added to the total value of the state. Table 2b illustrates another manner of weighting answers. A correct answer receives a weight of 1000. Predicting an earthquake when there isn't one only adds a weight of 500 to the model. Not predicting an earthquake when there is going to be one adds nothing to the model. This is a more subtle form of weighting since it not only allows a good and bad answer but also a not so good answer.



except perhaps for the use of the terms search limit and generation limit.

In order to prevent the evolution procedure from taking hours or days, it is constrained as to how long it can take. Each time an evolution cycle must occur, two variables are set. The search limit variable defines the number of times the parent may be serially mutated when searching for a better offspring. The generation limit variable defines the number of offspring to be generated.

The result is that a fixed number of offspring are generated. Further, each offspring can undergo a maximum series of mutations, since it is seldom that a single mutation results in a better model. To make the program more intelligent, one can increase these limit constants, but the result is a greater amount of computer time required between responses.

Again, for those adventurous programmers who want to make their programs even smarter, other methods can be employed to determine dynamically how many offspring and how many mutations are to be performed. There are other advanced evolutionary techniques that can also be employed. Interbreeding, majority logic and second order pattern recognition are just a

Figure 9: This is the main control logic of the simulated evolution technique. It first builds a dummy model, predicts an event, and receives the response. It then performs repeated versions and variations of the model looking for the optimum model to predict the correct answers.

few terms describing the advanced techniques available in computer science literature.

In Conclusion

If you have the patience, sit down with a pencil and paper and attempt to perform the process I've just described. Many mathematical books and papers contain random number tables you can use. As a result you should quickly see how this mathematical technique results in the creation of a model of some process. From there it is relatively simple to program.

Do not be afraid to try your own

improvements, but do so intelligently as you get a feel for the underlying processes. The process described herein is the basis for many computer versions of this artificial intelligence technique and has worked with varying degrees of success for a wide variety of goals. I might conclude from personal experience: Be careful how and to whom you expose this technique. There are many people who fear computers as they fear anything they don't understand and their enthusiasm for your creation may not match yours. With a good enough program model, however, you should be able to predict who these people will be. [*Hmm . . . our artificial life has defense mechanisms.*] ■

The following commentary was received from a reader and printed in the Technical Forum section of BYTE in the September 1977 issue.

On Finite State Machines and Their Uses

The two articles by M Wimble in the May and June 1977 BYTEs entitled "Artificial Intelligence. An Evolutionary Idea"¹ provided much food for thought for me, as well as many others I'm sure. If I had a personal computer, I'd try a crack at the earthquake prediction model.

However much I hate to dampen anyone's spirits, I must point out a basic limitation to the system that Wimble has outlined. The implementation is based on what is called a deterministic finite state automation. It's finite, since there are only so many states, and it's deterministic, since when one goes from one state to the next there is no ambiguity or uncertainty as to which state to go to next. Certain properties can be proved about what a finite state automation can do and what it can't do.

First, what it can do: recurrent pat-

terns are its forte. Given a recurrent pattern, the method Wimble outlines will construct a finite state automation that will predict what will happen next.

Now, what it can't do: the deterministic finite state automation cannot recognize or act upon all computer languages. You cannot use the method to teach your computer BASIC and expect it to digest your Star Trek program. The language level is of a higher sort than that which a finite state automation can recognize. A discussion of the four levels of grammar as outlined by Noam Chomsky, and the machines used to recognize sentences in those grammars would be helpful. I might suggest that there are more fruitful lines of endeavor (like perpetual motion machines or faster than light travel) than teaching a deterministic finite state machine a language.

Gerald Owens
POB 9038

Tucson AZ 85720

¹pages 9 and 15 in this edition.

Natural Language Processing and Small Systems

Harry Tennant

Introduction

What can possibly be said about the use of natural languages, that is the languages people use, with small systems? Where research is done on natural language processing, it is done on the largest computers available. To many computer scientists, the problem of enabling computers to understand natural languages at a reasonable level of competence is beyond the current technology. Consider what are probably the two best natural language processors yet produced by computer scientists: William Woods' LUNAR system which answered questions about rocks brought back from the moon, and Terry Winograd's system which manipulated blocks on a table in response to English commands; both are quite large programs. LUNAR uses one task with 256 K 36 bit words to discover the meaning of the user's query, then uses another task of 256 K words to answer the question. One question could take from three to 20 seconds to answer. Winograd's system did not need the quantity of data that Woods' system needed, but it still required 60 K words of 36 bits to operate in its limited world, consisting of a few blocks on a table. These are just two examples of the many natural language processing projects which have been conducted in recent years. These two (from the early 1970s) and nearly all the others since then share the property that they are large projects done on large machines using large amounts of memory. So what can possibly be said about natural language

processing and small systems?

The small system user is severely limited: he or she has comparatively little memory to work with, few languages to choose from (and those languages are not particularly suited to the needs of natural language processing), and usually few aids to software development, such as secondary storage, editing facilities, and debugging facilities. But among small systems users, there is a growing interest in the application areas of artificial intelligence: intelligent game playing, math, science and engineering aids, robotics, and natural language processing. In this article the general problems of computer based understanding of natural language are discussed briefly, and a few techniques that can be used on small systems to do a limited amount of natural language processing are presented.

Attempts have been made since nearly the dawn of computer history to make it possible for computers to understand the languages of people. It began as translation between natural languages, for example, from Russian to English. That kind of work was not successful. Later, research moved into the areas of natural language query of data bases and the study of the structure of human thought and memory through the modelling of human language behavior on computers. This is the work that is being done today. It looks promising, but it is still too early to tell if the work will actually provide users with the ability to communicate their thoughts to computers as efficiently as humans communicate with one another.

Natural Language Understanding on Computers

A conversation between two humans could proceed something like this:

Sam: Joe, how's your micro coming?

Joe: OK. I've moved on to the video cards.

Sam: When did you finish the cassette interface?

Joe: Last week. It took long enough for the chip to come.

Sam: Yeah, that's why I always deal with the fastest companies.

Joe: What do you think of the new video timing generator?

Sam: It will save some board space and maybe some money, too.

We do not know much about Joe and Sam, but we do know from this conversation that they both know something about microprocessors. We know that Joe is building one and that Sam knows quite a bit about it. When asked how his micro is coming, Joe thought of his computer, the problems he's had on it, the last section that he has been working on, the sections that have already been built . . . in other words, a great deal of information about his computer came to mind. Joe then thought of what Sam knew of the computer, and chose a relevant piece of information that Joe thought Sam did not know, and said that he was working on the video section. Now, Sam knows a lot about the computer, too. He is thinking about it just as Joe is. And so the conversation proceeds. Both Joe and Sam know a great deal about the computer. Both know about the problems of building a microprocessor, parts availability, etc. Their conversation is short, it uses few words, but it manipulates very large information structures in each of their minds as the conversation takes place. The conversation is not just a trickle of words between Joe and Sam, but it is mainly an activity inside their brains involving a great deal of information. The trickle of words is not what is really going on, it just triggers what is going on. The real activity is happening in the minds of Joe and Sam.

Now, what about a conversation with a computer? If it were to happen as the conversation above did, the computer would need to know a lot about the microcomputer that is being built. In other words, the computer would need knowledge very much like Joe's and Sam's. It would have to have some way of representing information about microcomputers: what they are made of; how they are built; the particular microcomputer being discussed; its state of completion; and so on. In addition to this, there must be some way for the computer to dis-

cover what the words in the conversation are referring to. How does it know that the conversation is about microcomputers, for instance? The word "micro" could refer to a microbiology program. As if that were not enough, let's say that the computer did interpret the first question correctly, did have information about the microcomputer, and decided on something to tell the questioner. It then has the task of presenting the information to the questioner in a form that he will understand. Add to this problem that humans converse on a wide range of topics, and learn about new topics without even trying, and the problem of enabling a computer to converse like a human becomes a large problem indeed. A full solution to the problem is a long way off, and quite possibly will require hardware beyond what is available today (a HAL 9000, perhaps?). But there are many steps toward natural language processing that can be done without a HAL 9000 and without 30 years of research and development.

As mentioned above, there are three main problem areas in natural language processing:

1. Representation of knowledge
2. Associating words with ideas
3. Presenting ideas

The problem of presentation of ideas by a computer will not be considered explicitly. Representation of knowledge and associating words with ideas will be considered in the next sections.

Representation of Knowledge

Before we approach the problem of representing knowledge on a computer, it may help to decide how to represent knowledge on a piece of paper. The first thing to decide is exactly what we want to represent.

Consider the microcomputer systems in figure 1. If we want to be able to converse with a computer about such systems, we need some way of storing what is known about them. Some of the things we know about System 1 are:

1. It has an 8080 microprocessor.
2. It has 1 K of read only memory.
3. It has 2 K of programmable memory.
4. Its only output is lights.
5. Its only input is a keyboard.

We know the following about System 2:

1. It has an 8080 microprocessor.
2. It has 1 K of read only memory and 1 K of programmable memory.
3. It has a scale as an input device (it is a digital scale).
4. It has a small keyboard as input.

5. It has as output a display of decimal digits.

The system can be represented hierarchically (in outline form) as:

System 1

1. Processor
8080
2. Memory
1 K read only memory
2 K programmable memory
3. Output
Lights
4. Input
Keyboard

System 2

1. Processor
8080
2. Memory
1 K read only memory
1 K programmable memory
3. Input
Scale
Keyboard
4. Output
Decimal display

Now suppose we want to use this data base to answer questions about System 1 and System 2. (Note that a data base like this could be extended to include many other similar systems. It can be extended to as many as memory will allow. But, consider how trivial it is in detail and breadth compared to a human's knowledge. For this reason, one cannot expect the computer to respond with anything like the intelligence of a human. The relevant questions are 1) is there enough data in the data base to make it worth doing and, 2) is the English interface to the data good enough to warrant making it.) When asked what the processor is for System 1, we consult our outline of System 1, look under "Processor," and reply that it is an 8080. When asked what the output for System 2 is, we consult the outline for System 2, look under the output to find "Decimal display," and we return that. When asked about the inputs to System 2, we respond that they are a scale and a keyboard. But note that a more complicated problem arises if we ask what kind of processor is in the system that uses the scale. First we must find all systems that use scales (we may not even realize that a scale is an input device). For all the systems using scales, we must then find what their processors are, and return that information. This is not a problem if we have only two systems, as in the example. But if we had many systems whose outline descriptions filled many sheets of paper, searching all the descriptions for the ones that use scales could be a major effort. One of the advan-

tages of natural language is that items can be referred to by their descriptions instead of their names. We referred to System 2 not by its name, but by its description, ie: the system that uses a scale. Because this is such an important feature of natural language, it is very important to be able to deal with it. One way of doing so is to make a new set of outlines for each of the items mentioned in the original outlines. For example:

8080

1. Processor in
 1. System 1
 2. System 2
- 1 K Read only memory
 1. Memory in
 1. System 1
 2. System 2
- 1 K Programmable memory
 1. Memory in
 1. System 2
- 2 K Programmable memory
 1. Memory in
 1. System 1
- Lights
 1. Output of
 1. System 1
- Scale
 1. Input of
 1. System 2
- Keyboard
 1. Input of
 1. System 1
 2. System 2
- Decimal display
 1. Output of
 1. System 2

Figure 1: System 1 and System 2 diagrammed as components connected to a common bus.

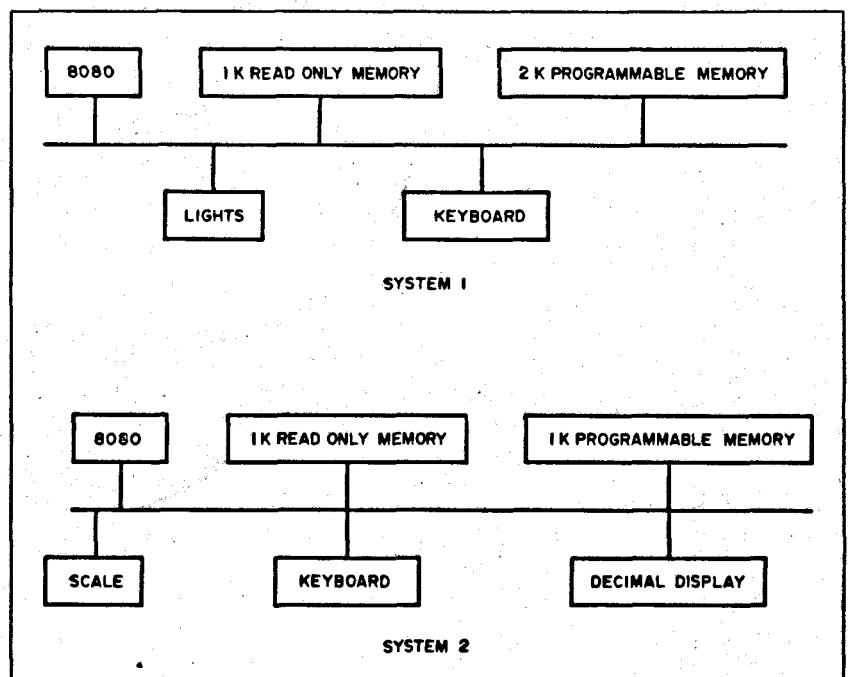
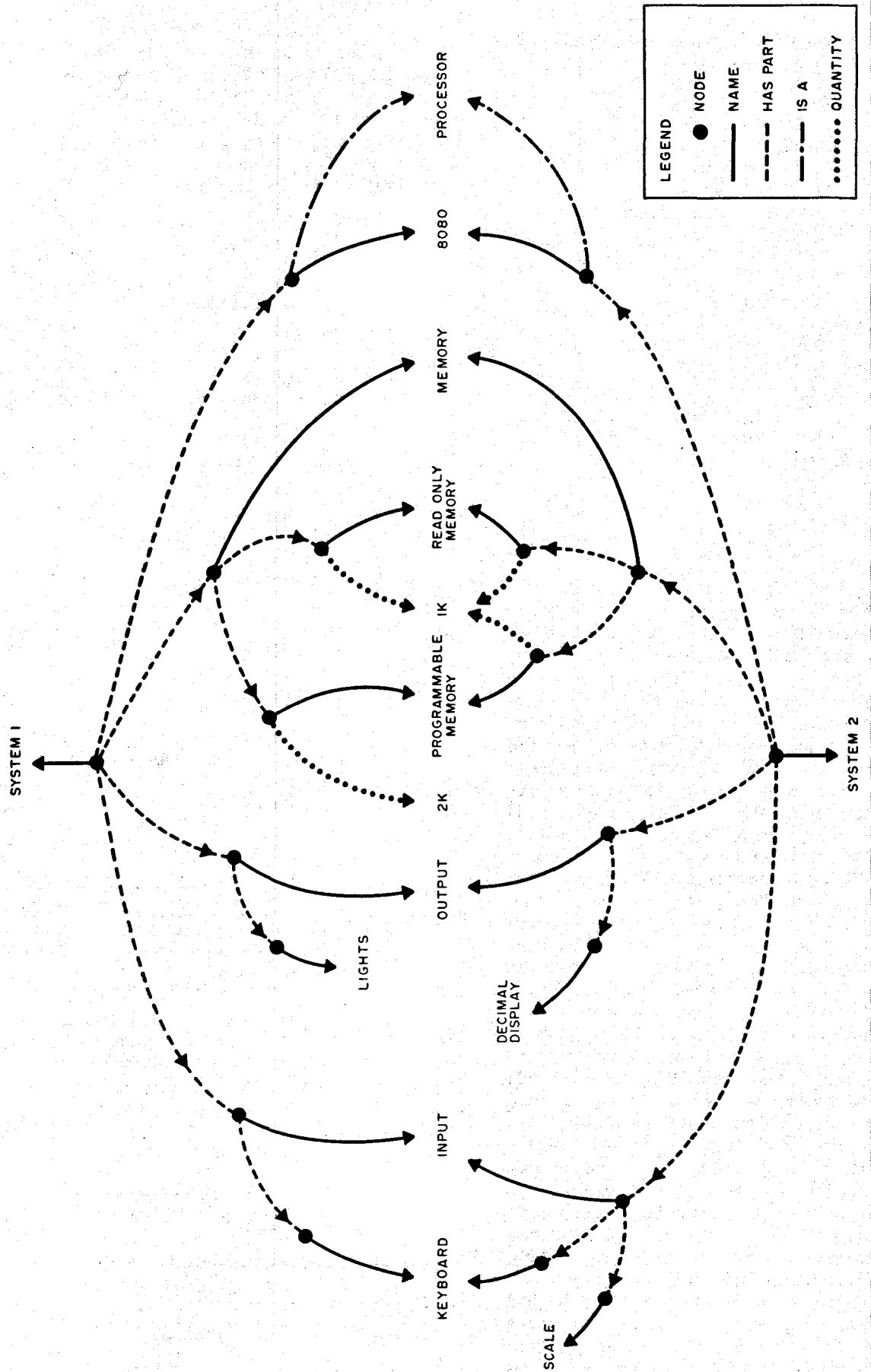


Figure 3: More complex semantic net. This semantic net represents the same systems as the net in figure 2, but in a more detailed fashion. In this net, each node represents an entity. For example, one of the parts of System 2 is an entity called the MEMORY. It is composed of two parts, the entity called the READ ONLY MEMORY and the entity called the PROGRAMMABLE MEMORY. As in the other semantic net, there is some flexibility in what to call the links and nodes. For example, "IS A" suggests that the 8080 is a member of the category of things called PROCESSORS. One could also say that a READ ONLY MEMORY is a member of a category of things called MEMORY, but in this net, READ ONLY MEMORY is represented as a part of a thing named MEMORY. A fine distinction, but it may be significant when the net is interpreted by a program. Notice that reverse links (ie: NAME OF, PART OF, QUANTITY OF, EXAMPLE OF) are not shown.



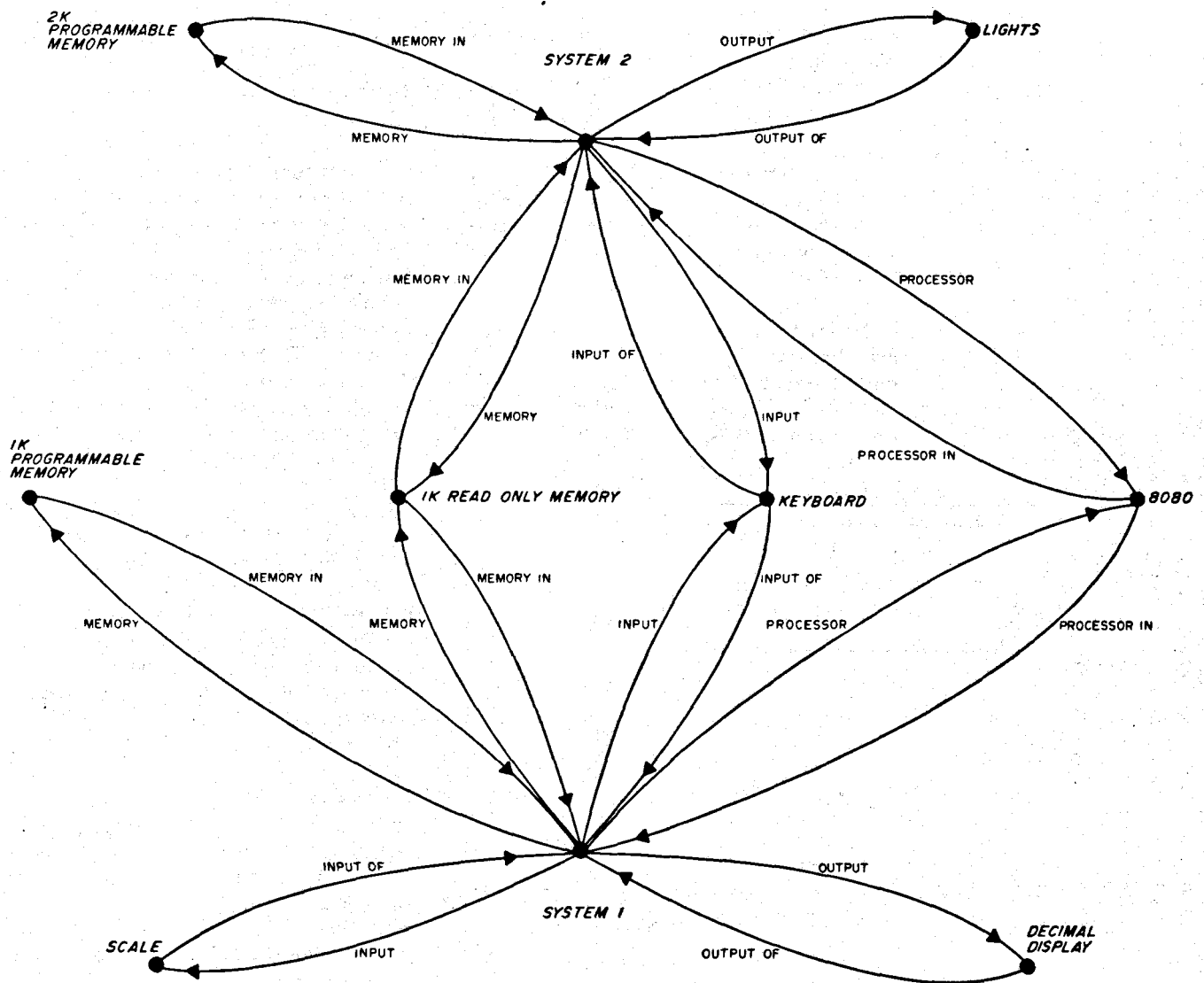


Figure 2: Simple semantic net. This net represents the connections of the components of System 1 and System 2. Each component is represented by a node. The nodes are related to one another through labelled links. Notice that the names chosen for the links and nodes are largely at the discretion of the net designer.

Now, think again about representing a large number of systems in this way. It will take more memory, but it will provide access to items such as scale or 8080. If a hundred systems are represented on sheets of paper, we would probably have an index or table of contents to direct us to the appropriate page to find the outline we seek, thus speeding up the response. I have not forgotten that in many small systems the trade-offs between memory and processing time often are in favor of saving memory space. What is presented here is intended to illustrate the trade-offs, not to make the decisions for the user.

It is easier to see this type of representation when it is presented in graphic form. The representation used in figure 2 is called a semantic net. Each outline is represented

by a node and the relationships between nodes are represented by directed labelled arcs. The idea of a semantic net is that nodes are entities and arcs are relations between entities. The net shown in figure 2 is not the way most natural language researchers would represent the information about our two systems. For example, "System 2" is the name of something, it is not the thing itself. An 8080 is a part of the thing whose name is "System 1," and that thing that is called an 8080 has the function of "Processor" in the thing called "System 1." A diagram of representation like this is shown in figure 3. It is more explicit and more correct than the representation of figure 2, but is harder to build, more difficult to interpret, and requires more memory to represent.

Semantic nets, in whatever form one

prefers, are a flexible and easily accessed way to represent knowledge. They simulate associative memory, as humans seem to have. But how are they represented on a computer? The answer to this comes in three parts:

1. Outlines can be represented by lists.
2. Lists can be represented on computers.
3. The index or table of contents to outlines can be represented as a hashed table.

Nodes can be represented in lists by having alternate elements be arc labels and related nodes. We must first reduce each multiple word node name and arc label to single word names, then just list them as follows:

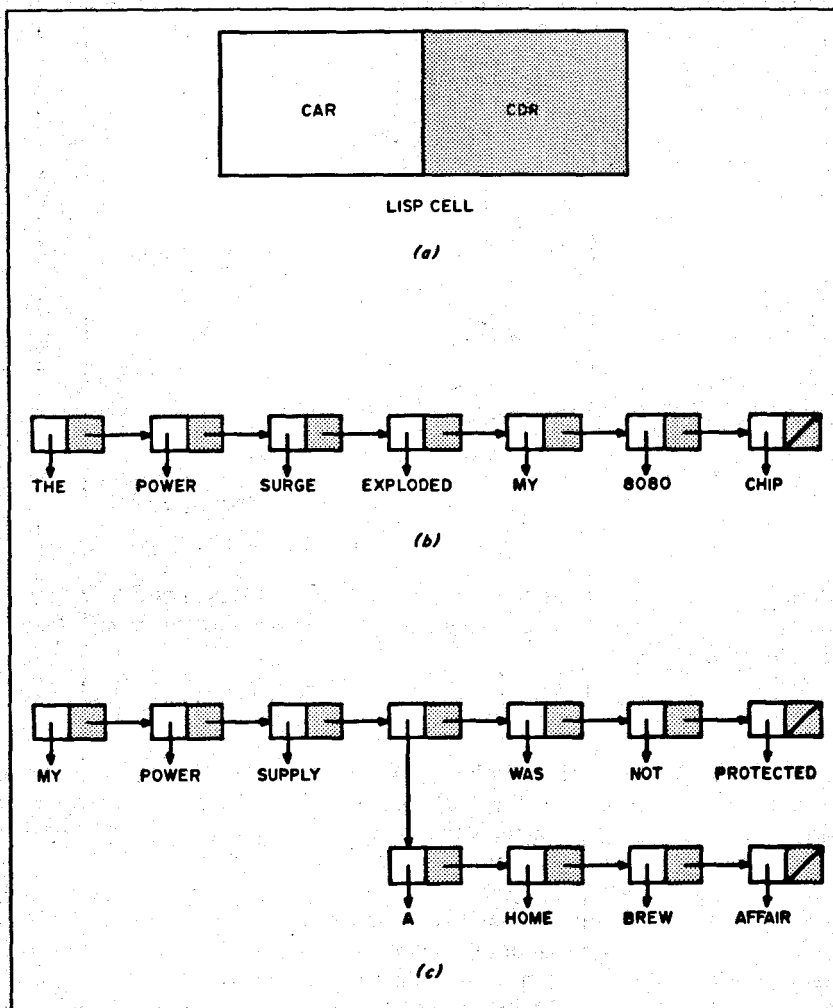


Figure 4: LISP cells. In (a) a LISP cell is shown. In an actual implementation there may be some additional bits in the CAR and CDR to carry information about how the cell is used (whether it is an atom, for instance). In (b) the list of words (THE POWER SURGE EXPLODED MY 8080 CHIP) is shown as a string of cells. The pointers that point to the words are actually pointing to the cells that represent those words (see figure 5). In (c) the cell representation of an embedded list is shown. The list is (MY POWER SUPPLY (A HOME BREW AFFAIR) WAS NOT PROTECTED).

8080 (Processor-in (System1 System2))
 System 1 (Processor (8080) Memory (1 K Read only memory, 1 K Programmable memory) Output (Lights) Input (Key-board))
 Keyboard (Input-of (System1 System2))

Most natural language projects are written in LISP because of its ability to handle list processing better than most other languages. Lists are represented in LISP in the following way: A list is composed of cells. Each cell has two parts, as shown in figure 4a. The two parts are called the CAR and the CDR of the cell. Each entity (node in the semantic net) is represented by a unique cell. A list of words would be represented by a string of cells. The CAR of each cell points to a node cell and the CDR points to the next cell in the string. The CDR of the last word has an end of list marker in it. A list of words ("The power surge exploded my 8080 chip") is shown in figure 4b.

A list can be represented within another list by having a CAR point to the first element of the inside list, instead of pointing to a node cell. A list within a list (My power supply (a home brew affair) was not protected) is shown in figure 4c.

The index table for finding node names is provided automatically in LISP. The table is called the OBLIST or OBARRAY and the lists are called PROPERTY LISTS. If you do not happen to have LISP, the index table can be built as a hashed table of node cells with associated pointers to the property list associated with that name. A diagram representative of the whole configuration is shown in figure 5.

If the cell is a node cell, the CAR points to a location where the name of the cell (the character string) is held. The CDR points to another cell which is the beginning of the property list of the node. In cells that are not node cells, CARs and CDRs are both just addresses that point to other cells. (One exception is the cells that represent the relationships between nodes. These will also be on the OBLIST as "node cells," but their CDRs will contain end of list markers instead of pointers to property lists.) In LISP, a special bit is set to designate whether a cell is a node cell (called an ATOM in LISP) or just a regular cell. The amount of memory that needs to be addressed by the CAR and CDR of each cell determines the number of bytes that each cell must be composed of.

Associating Words with Ideas

The process of understanding natural languages certainly has something to do with

associating words with ideas. It should be stressed from the beginning, however, that when humans understand something it is a process that uses much more than just word definitions. The process of understanding in humans involves interpreting the words they hear or see with the various meanings for those words, all viewed in the context of the current conversation, the environment of the conversationalists, and other factors. For example, the sentence, "I'll take a pancake," can be assumed to mean very different things depending on whether it is heard in Uncle John's Pancake House, or in a store that sells fans for relay racks (a pancake fan), or if it is said at a cosmetic counter (pancake makeup). The various conflicting meanings of "pancake" do not even occur to the people in question. A waitress would be unique indeed if she asked her customer if she preferred the pancake on a plate or on her face!

The problem of multiple meanings, contexts and other details of understanding will be ignored for the time being. In a small system there are limits to what can be done linguistically (in this respect, all contemporary systems seem small!). But a step toward natural language can be taken, however small. The goal we will assume is that an inexperienced user will be able to address the natural language processor in language that the user is most fluent in, and that the language processor will respond in a manner that the user finds appropriate.

We will be considering a situation in which natural language is being used as an interface language between a user and either procedures or data in a computer. The user types a sentence and the computer interprets the sentence and does what it is understood (by the computer) to mean. For example, say the computer holds a data base about various microprocessor systems like the one that was described above. The user asks questions about the systems and the computer provides answers.

Keywords

The simplest method of interpreting a sentence is to look for particular words, called keywords. If a keyword is found, a response is output. For our system, a useful set of keywords would be the names of all the nodes in the semantic net. For a response the system could print the property list which represents how that node is related to other nodes in the net. For example:

User: Tell me all about system1
 Computer: System1
 Processor
 8080

Memory
 1 K Read only memory
 1 K Programmable memory
 Output
 Lights
 Input
 Keyboard
 User: What information do you have on lights
 Computer: Lights
 Output-of
 System1

These responses would be quite appropriate for the questions asked. Major improvements

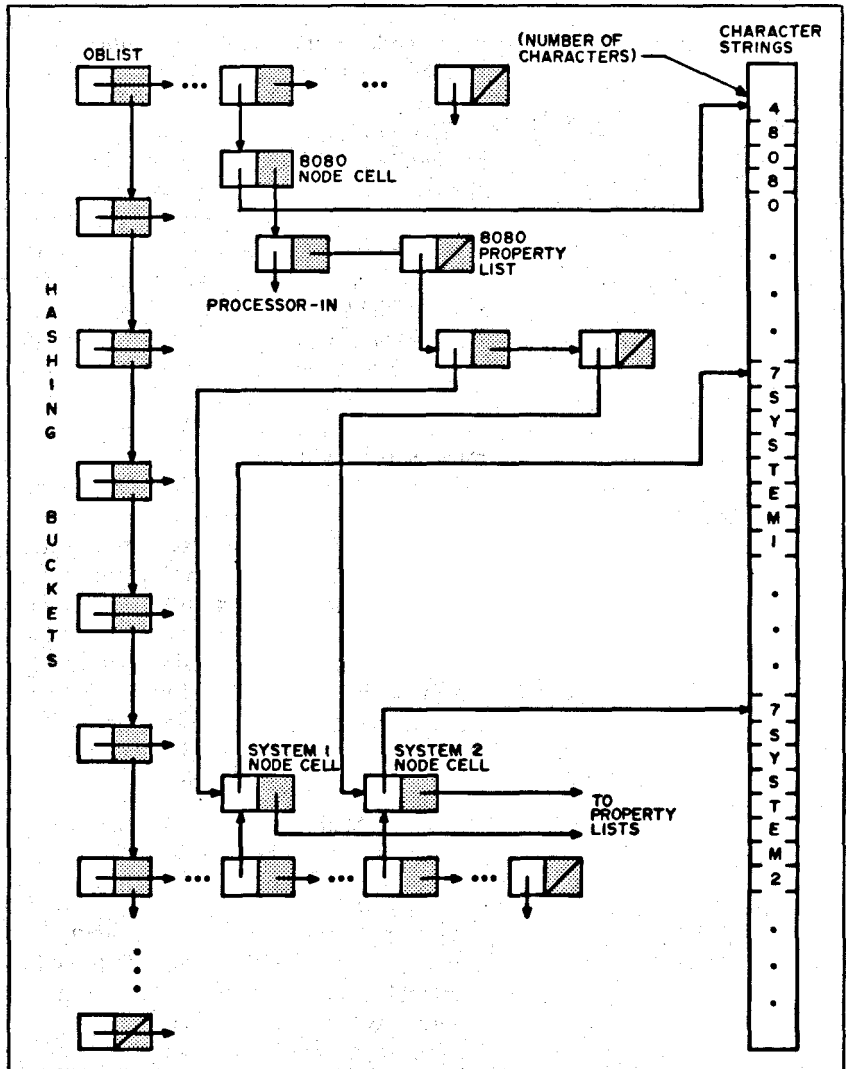


Figure 5: Representation of atoms. Each node in the system is represented by a unique cell, here called a node cell. In this figure the node cells for three nodes are shown, for 8080, SYSTEM1, and SYSTEM2. The CAR of each node cell points to a place in memory where the character string of the name of the cell is stored. The CDR of each node cell points to the property list of the node. The property list of the 8080 node is shown. The pointers to words shown in figure 4 are actually pointers to the node cells of the words. All node cells are chained together by the OBLIST. The character string names of the nodes are stored in a part of memory that has not been divided into cells. The CARs of the node cells point to an address that specifies the number of the characters that follow that are included in the character string of the name of the node. Links have cells on the OBLIST, just as nodes do. The only difference is that links do not need property lists. The pointer to Processor-in would actually be a pointer to the cell that represents Processor-in.

can be had by a few simple changes, however. First, what would happen to "Tell me what you know about keyboards"? The language processor would not recognize keyboards as the same as "keyboard", so no information would be found. The easiest way around this problem is to strip the Ss off all the node names when building the semantic net, then strip them off all the words input by the user. However, this procedure runs into problems for words that end in S, like "process". Another problem is words ending in "es", like "processes". Actually, there are algorithms for analyzing word endings that correctly reduce words to their roots for nearly all the special cases like these.

The whole problem of word endings can be avoided by using a universal character. A universal character is one that matches all other characters. If # were a universal character, the node names could be written as "light#" and "process#". These would match "light" and "lights", "process" and "processes". Tricks like this can help, but may produce problems by also matching "lightning" and "lighter", "processor" and "processing". Therefore, universal characters must be used with care.

Another improvement deals with nodes that have multiple word names. A user would probably ask about "system 1" instead of "system1". Multiple word names can be handled by more property lists, one for each first word in multiple word names. These property lists would contain a list of the words of the multiple word name, followed by the corresponding node name in the semantic net. For example, system1 and system2 could be referred to by:

```
System (Mult-wrd-names ((1) System1 (One)
System1 (2) System2)).
```

This property list signals the language processor that "system 1", "system one" and "system1" are all to be interpreted as "system1". Also, "system 2" means the same as "system2".

The same mechanism can be used to allow synonyms. System 1 may be affectionately known as the "Bit Byter", "Old Smokey" (for its power supply problems), or "Zapper". These names can be interpreted as the same as "system 1" with the following property lists:

```
Bit (mult-wrd-names ((Byter) System1))
Old (Mult-wrd-names ((Smokey) System1))
Zapper (Mult-wrd-names (() System1)).
```

Just as universal characters can be used, universal words and phrases can simplify specifying a large set of synonyms. For example, the multiple word name (President # Washington) would match all phrases that

begin with "President" and end with "Washington". This would match (President Washington), (President George Washington), (President G Washington and also (President Ford never met George Washington).

A relevant question is: What good is putting the keyword in the middle of a sentence? Why not just have the user type keywords and forget about the rest of the sentence? There probably are few if any good reasons for trying to create an illusion of natural language understanding in this way other than that it is a fun trick.

Nodes and Links

The keyword approach to natural language processing is imprecise, and so it is prone to many errors and misinterpretations. There is an approach that is somewhat more precise, and allows correct interpretation of much more complex sentences without being much more complicated. This method involves identifying both node names and link names from the semantic net, then combining them to print only the parts of the semantic net desired by the user. The link names in our example are processor, memory, input, output, processor in, memory in, input of, and output of. There will usually be many fewer link names than node names. A user's sentence is processed by collecting node names, then link names, as in:

Give me the Old Smokey processor and output.

Using a previously mentioned definition this is seen by the processor as:

```
xxx xxx xxxx system1 processor xxx output
              (node) (link)          (link)
```

to which the appropriate response is:

```
System1
  Processor
    8080
  Output
    Lights.
```

The response is formed by searching the property list of the node mentioned for each link named. Then the node name, the link name, and the names of the nodes pointed to by the link are printed in outline format.

If link names are found to the left of node names in the sentence, the inverse link names must be substituted. For example,

What are the input of and processor in system 2

is seen as:

```
xxxxx xxx xxx input-of xxx processor-in system2
              (link)          (link)          (node).
```

The reverse links are used, and the sentence is reinterpreted as:

system2 input processor.

The property list of system2 is checked for input and processor links and the following is printed:

```
System2
  Input
    Scale
    Keyboard
  Processor
    8080.
```

This process is diagrammed in figure 6. This technique will allow quite complicated sentences to be interpreted if synonyms are chosen judiciously for nodes and links. Users on a natural language processing system tend to use very short and incomplete sentences if they can, which is also allowed with this system.

Conclusions

There is naturally a lot that keyword based systems cannot do that humans can. For instance, keyword systems cannot understand pronouns, or when to use different word meanings (like pancake). Humans have little trouble using these. The difference is, of course, in all the information that keyword systems throw out when they disregard all words that are not keywords. Also, words carry with them more than just a definition. Most words say things about the words that are surrounding them. The word "the" says that the word to its right is either a noun or a noun modifier. In the system described above, the keyword "input of" tells us that either some kind of input device has preceded this phrase in the sentence, or a reference to a system will follow it ("what is the input of system 2"), or both ("a keyboard is input of system 2").

Natural language research today has moved far beyond the realm of keyword analysis to use not only knowledge about words, but knowledge about the things and events that the words refer to, and knowledge about the way text and conversations are structured. In a data base like the microcomputer data base used here, the system would retain information about the components that every system would probably have, the probable uses of systems, more detailed information about each of the components, and so on. The information that a human uses when discussing microcomputers would be collected and added to the knowledge base to be used when the computer is discussing microcomputers. This information is grouped into

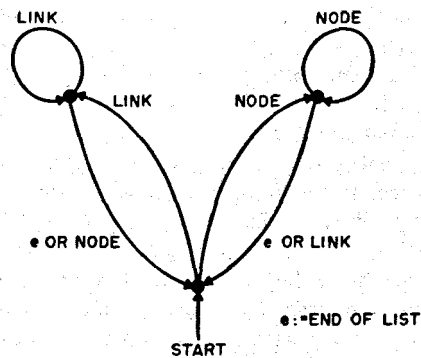


Figure 6: Node and link interpretation. This simplified method of interpreting which nodes and links mentioned in a sentence should be associated will often assume the proper interpretation to the input sentence. However, node and link names and synonyms must be chosen with care. A more successful method requires greater attention to words other than those in node and link names, particularly conjunctions, prepositions and relative pronouns.

collections and the collections are associated with the concepts they describe.

It is nearly impossible to discuss how well such a system can work. We do not yet have any kind of scale for measuring natural language performance if it falls into the subhuman range. The only really useful measure is whether or not it does what it is supposed to. Unfortunately, that too is difficult to ascertain. I have implemented a system like the link and node system described here, but it was quite a bit more complex. It attempted to account for every word in the sentence in order to prevent misinterpretations. It handled pronouns fairly successfully and was good at identifying items by their description. It responded to the user in full sentences and in outline form. It also required a large system to run on, primarily because it had a fairly large data base. With all this, it still usually took a new user several examples before he could communicate with the system in a useful way.

With this experience, what do I think natural language systems can do on small systems? The question is vague, so the answer is vague: a little bit, but not too much. That not-very-helpful answer means that one must decide why he or she wants a natural language processor, then consider the techniques described in this article to decide if they will meet his or her needs. These techniques must then be compared to nonnatural language techniques.

The benefit of the language understanding techniques described here is primarily based on the power of the semantic net

representation. Semantic nets have one advantage over other representations in that concepts are associated in a way similar to the way they seem to be associated in human memory. An important aspect of using natural language as an interaction language between humans and computers is that, if it works, it allows the user to state his requests in the same way that he thinks about his requests. A user using natural language to interact with a computer is manipulating an enormous amount of information in his or her mind, encoding a small part of that knowledge into the words of the conversation, assuming that the listener (the computer) can use these few words to manipulate its large informa-

tion structures. On a small system, the information structures that the computer has to manipulate can not be nearly as large as the human's. The words of the conversation can be associated with ideas, but not on the lavish scale of association available to humans. Finally, most of the words of the conversation are thrown away using the techniques that a small system can support. What natural language processing can be done on small systems? Not enough to be able to compare it to natural language processing in humans, but perhaps enough to allow a user to learn to communicate effectively with the computer in a way that is close to the way the human brain thinks: through associations and descriptions.■

SIMULATION OF MOTION

Simulation of Motion:

Part 1: An Improved Lunar Lander Algorithm

Stephen Smith

One of the most delightful applications for personal computers is games, not just playing them, but creating them. If you are like most enthusiasts, you will have begun with random number games like blackjack, but sooner or later you will want to work with games involving moving objects. To describe that motion using a microcomputer you will need to use a form of simulation. The simulation could involve detailed mathematical models solved with elegant numerical techniques.

More likely, the novice will begin by following the pattern of the simple lunar lander games which have appeared often in BYTE (see "Kim Goes to the Moon," by Butterfield in April 1977 BYTE, or "Controlling Small DC Motors with Analog Signals" by Dwyer, Critchfield and Sweer in September 1977 BYTE). The truly advanced simulations are best left to professionals with mainframe computer power, but the home user can progress well beyond the simple lunar lander game. By picking up the basic physics and simple numerical methods presented in this article and the following ones, you will learn to simulate a wide variety of motion. Whether you use these simulations to create games, like the real time LEM simulator presented here, or to develop new applications for your per-

sonal computer system, you will acquire some valuable additions to your applications software toolbox.

For any application involving motion, your simulation will be required to predict the speed and position of an object at some time in the future. The predictions can be made using a microcomputer if you first limit the type of motions considered at any point in the program. In the lunar lander game, for example, the excursion module (LEM) is only allowed to move up and down. The simulation is said to have one degree of freedom. Other degrees are possible, but the separation into different degrees of freedom is an important first step.

Let's see how a one degree of freedom simulation is performed. Thanks to Sir Isaac Newton and his apple (that was a fruit, not a computer), we know that an object will continue to move in any degree of freedom without changing speed until a force acts on it. To predict how the LEM will move, we need only to examine the forces which might be present and determine how they effect the up and down motion.

Because the moon has no atmosphere to involve us in aerodynamics, only two forces need be considered, gravity and thrust. Gravity makes the LEM fall faster. Thrust

Generic Unit	Metric	
Length	1 meter	= 3.2808 feet
Velocity	1 meter per second	= 3.2808 feet per second
		= 2.2369 miles per hour
Acceleration	1 meter per second per second	= 3.2808 feet per second per second
Mass	1 kilogram	= 2.2046 pounds (mass)
		= 0.0685 slugs (mass)
Force	1 newton	= 0.2248 pounds (force)

Table 1: This article was written using the metric system of units. As the front runners in an exciting new technical hobby, we should be more ready than most to accept the coming metric conversion in this country, but if you haven't been converted yet, the above table will be useful.

makes it fall more slowly. The exact effect of each can be calculated with only a few operations.

Gravity is the simpler of the two. It has exactly the same effect on every object. During each second of a lunar landing near the moon's surface, the moon's gravity will make a LEM fall 1.62 meters per second faster. (Those of you who wish to land on more exotic heavenly bodies are referred to table 2.) In most simulations, speed and position are considered positive if they are directed upward, in this case away from the lunar surface. To simulate 1 second of fall through lunar gravity we must subtract 1.62 meters per second from the present speed. If the LEM is moving at -100 meters per second now (100 m/sec downward), 1 second later it will be moving at -101.62 meters per second.

In many games, the effect of thrust is also simulated by a constant change in speed. Often it is given in multiples of gravity called "g"s. One "g" of thrust adds 1.62 meters per second to the speed, just as gravity subtracts that amount. Two "g"s add twice that, and so on. This assumption reduces the complexity of the simulation, but it fails to demonstrate the way in which forces actually cause changes in speed.

Heavenly Body	Surface Gravity (m/sec ²)	Heavenly Body	Surface Gravity (m/sec ²)
Moon	1.62	Asteroids	
Earth	9.80	Ceres	0.85
Mercury	3.95	Pallas	0.54
Venus	8.72	Juno	0.21
Mars	3.84	Vesta	0.43
Jupiter	23.16	Jupiter's moons	
Saturn	8.77	Ganymede	3.43
Uranus	9.46	Io	2.26
Neptune	13.66	Europa	1.98
Pluto	4.89	Callisto	3.20

Note that the gravitational accelerations shown in this table are surface accelerations, valid during the final stages of a landing when a spacecraft is relatively near the heavenly body. A more complicated simulation is required if movement far away from the heavenly body is contemplated.

Table 2: Players who grow adept at lunar landings may wish to try landing on some other heavenly bodies. The above table of accelerations due to gravity is provided for them. .

Unlike gravity, forces such as thrust do not have the same effect on every object. They have a larger effect on light objects than they have on heavier ones. It is important to consider this fact in accurate simulations, because weights can change. The LEM becomes lighter as it burns fuel to create thrust. A given value of thrust will have a larger effect toward the end of the flight than it will at the beginning.

Weight is not really the correct term to use when calculating that effect. We should talk instead of mass. The difference is subtle, but important. Mass is a basic property of matter. Weight is the result of gravity pulling on the mass. A man on the moon weighs only 1/5 as much as he does on earth, but his mass is the same. This is true because the moon's gravity pulls only 1/5 as strongly on his mass. The effect of a force is determined by the mass of an object, not by its weight. A given thrust will have the same effect on a LEM whether the LEM is landing on the moon, on earth, or is floating "weightless" in space.

In the metric system, the unit of mass is the kilogram. The unit of force is the newton. These units are very convenient for calculating the effect of a force on the motion of an object. The force (in newtons) divided by the mass (in kilograms) is exactly equal to the rate of change in speed ("acceleration" in meters per second per second). No additional constants are needed as they are when units of feet and pounds are used. For example, let our LEM have a mass of 1000 kg and let its engine produce a thrust of 10,000 newtons. To simulate 1 second of thrust, a program would add 10 meters per second to the speed (10000/1000) to account for 1 second's worth of acceleration.

Remember though that during the same second 1.62 meters per second must be subtracted to simulate the effect of gravitational acceleration. The actual change in speed will be 10.-1.62=8.38 meters per second. In two seconds, the change will be twice that or 16.76 meters per second. In half a second, the change will be one half as much and so on. While this may seem obvious, it illustrates an important point. The change that each force makes in the speed in 1 second may be determined separately. The separate effects are added up and then multiplied by the length of time we are simulating to find the actual value the simulation program will add to the speed.

Now that we can predict speed, let's apply the same technique to predict the position. We have shown that if the LEM is moving downward at 100 meters per second now, (speed=-100) then in 2 seconds the speed will be -100.+2.x(THRUST/MASS

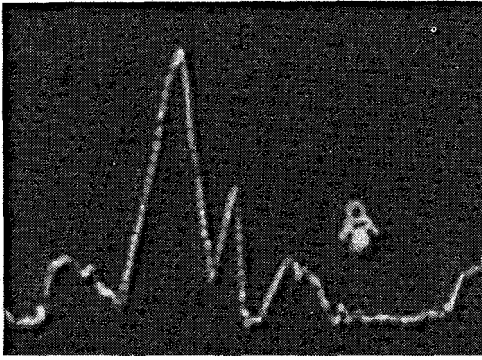


Photo 1: A scene from the "lunar lander" program which is the Digital Equipment Corporation's graphics equipment demonstration program. This simulation is a real time model of a lunar landing in which a light pen is used to input control information and displays track the landing. The object of the game is to land near (but not on) the only MacDonalds' hamburger stand on the moon. This simulation, like the one discussed in the article, has two degrees of freedom; superficially it differs from the program of this article largely in its incorporation of real time graphic display light pen control inputs and a model of the lunar terrain.

-1.62). Similarly, if the LEM is 10000 meters above the moon now, in 2 seconds it will be $10000 + 2 \times (\text{speed})$ meters up. Just as we multiply the forces by time and add the product to the speed, we multiply the speed by time, and add the product to the position.

What we have just done is to predict the speed and position at a "step" of 2 seconds into the future. In the jargon of simulation, 2 seconds is the step size. The step size can take any value you choose. Returning to the 1000 kg LEM, let the step size be 0.1 seconds. For a present speed of -100 meters per second, the speed predicted for 0.1 seconds in the future is $-100 + 0.1 \times (10000 / 1000 - 1.62) = -99.16$ meters per second. If the position now is 10000 meters, then the position predicted for 0.1 seconds in the future is $10000 + 0.1 \times (-99.16) = 9990.08$ meters above the moon.

Using these values of speed and position we can find new values for the forces and mass. We can then step the simulation into the future once again. The process can continue indefinitely, but usually one or more variables is tested for an end condition at each step. The test might be on position (Are you still above the moon?), on mass (Is there fuel remaining?), or on some other variable. Should any of the tests fail, the program will branch and end the simulation.

Adding a New Degree of Freedom

You know the basic procedure for simulating motion in one degree of freedom. The LEM simulation has been in one degree because we have only predicted the up and down movements. These are called vertical motions. Suppose that we also predict the way the LEM moves horizontally, in other words, from side to side. The pilot must not only reach the surface of the moon successfully, but also land close to his target. While the pilot's task has become more complicated, our simulation fortunately has not. Just as we are able to calculate the effects

of each force separately, we are able to make calculations for speed and position separately in each degree of freedom.

To make those calculations for the second degree of freedom, first determine what forces are acting. Gravity, by definition, acts only up and down. It does not enter into the horizontal calculations. So far, thrust has also been limited to vertical action, but we can easily add a second thrust acting to the side. Positive horizontal thrust should cause the LEM to move left, while negative thrust moves it right.

Since there are no other forces to consider, the change in horizontal velocity (in meters per second) will be exactly equal to the horizontal thrust (in newtons) divided by the mass (in kilograms). This is, of course, the same equation used in the first or vertical degree of freedom. Similarly, the same equations used to calculate vertical speed and position will be used to calculate horizontal speed and position.

Return to the example used earlier, but also consider the horizontal motion. Let the LEM start 100 meters to the left of its target moving at 10 meters per second to the right. Generally motion to the left will be considered positive and to the right negative, so the horizontal speed is -10 meters per second. We found that during a step of 0.1 seconds the vertical speed changed from -100 to -99.16, and the position changed from 10000 to 9990.08. Quite apart from those calculations, we may set a horizontal thrust, say 5000 newtons, and find that during the same step the horizontal speed will become $-10 + 0.1 \times (5000 / 1000)$ or -9.5 meters per second. The horizontal position will become $100 + 0.1 \times (-9.5) = 99.05$ meters. After making these calculations, the simulation program should check for end conditions and, if none are found, begin another step.

The speed of a computer makes it possible to find results quickly for times far into the future, even if the step size is quite small. This is fortunate, because as our simu-

Listing 1: Most of the lunar landing games I have seen are not flexible enough to run a two degree of freedom real time simulation as described in this article, so I have included this listing. At each initialization, this program finds a random set of starting conditions (speed, position, mass, etc) that is consistent with a safe landing. It then keeps track of speed, position and fuel consumption, printing them as required, and indicating when the surface has been reached. The following adjustments will have to be made by each user:

1. Function *USR(X)* must be provided to return the current desired thrust settings, 0 to 100% in the vertical direction, and -100 to +100% in the horizontal direction. These inputs are best achieved by analog to digital conversion from joysticks or slide pots.
2. The step size and print interval must be adjusted for your system clock and peripheral speed in order to simulate real time operation accurately.
3. Function *RND(1.)* is assumed by the program to return values between 0. and 1. Alterations may be necessary to suit your version of BASIC.
4. The comments printed by the program have deliberately been kept short. A better game could be fashioned by adding instructions, comments on performance, low fuel warning, etc. In other words, customize the simulation to suit your own tastes.

lation stands now, an error is introduced at each step that becomes worse as the step size becomes larger. The error occurs because in a real LEM the mass and speed are changing all the time, but in our simulation they can be changed only between steps. A variety of numerical methods have been developed to cope with this problem. In our example, simply using the average of the beginning and ending values in each step would be quite effective. Actually, if the step size is small, say 0.01 seconds, even this is not necessary. It is true that by using the average values the program could be made to run faster, but it would also need to store several extra variables, require more lines of code, and use more memory. Obviously, there are trade-offs to be made among speed, accuracy, complexity and size. In each simulation, the programmer must decide which combination is best.

For our games application, the combination is not critical. A high degree of accuracy is not required, and the program is short enough that memory requirements should

```

010 REM LUNAR LANDING SIMULATION
020 REM SET FUEL SAFETY FACTOR
025 REM ADJUST TO CONTROL DIFFICULTY
030 LET S=1.3
040 REM SET STEP SIZE AND PRINT INTERVAL
050 LET D=0.01
060 LET K=1.0
070 REM SET GRAVITY ACCELERATION
080 LET G=1.62
090 RANDOMIZE
100 REM SET NEW STARTING CONDITIONS
110 LET M=1024.+1024.*RND(1.)
115 PRINT "LEM MASS =",M
120 LET F=G*M*(4.+4.*RND(1.))
130 PRINT "MAX THRUST =",F
140 LET A=1.333*F/M-G
150 LET V=F/M*64.*RND(1.)
160 LET U=0.
170 LET Y=V**2./(2.*A)*(1.+RND(1.))
180 LET X=V
182 REM V IS VERTICAL SPEED
184 REM U IS HORIZONTAL SPEED
186 REM Y IS VERTICAL POSITION
188 REM X IS HORIZONTAL POSITION
190 REM HALF OF MASS IF FUEL
192 REM M-P IS FUEL REMAINING
200 LET P=M/2.
210 REM FIND FUEL BURN RATE, I
220 LET I=(2.*Y+V**2./G)/(1.+A/G)
230 LET I=P/(SQR(1/A)*F*S)
240 PRINT "ALTITUDE, SPEED, FUEL, RANGE"
250 REM BEGIN DECENT CALCULATIONS
260 PRINT Y, V, M-P, X
270 LET T=0.
280 IF M=P THEN 360
285 REM GET VERTICAL THRUST
290 LET A=USR(1.)*F/100.
300 REM GET HORIZONTAL THRUST
305 LET B=USR(2.)*F/100.
310 LET M=M-(A+B)*I*D
320 IF M>P THEN 360
330 PRINT "FUEL EXHAUSTED"
340 LET A=0.
342 LET B=0.
350 LET M=P
358 REM PREDICT NEW U, V, X, Y
360 LET V=V+D*(G-A/M)
370 LET U=U+D*B/M
380 LET Y=Y-V*D
390 LET X=X-U*D
395 REM TEST FOR END CONDITIONS
400 IF Y<4. THEN 440
410 T=T+D
420 IF T>K THEN 260
430 GOTO 290
440 PRINT "MODULE HAS LANDED"
450 PRINT "SPEED =",V
460 PRINT "RANGE =",X
470 IF V<3. THEN 500
480 PRINT "BETTER LUCK NEXT TIME"
490 GOTO 110
500 IF X<128. THEN 530
510 PRINT "IT'S A LONG WALK TO BASE"
520 GOTO 110
530 PRINT "CONGRATULATIONS, GOOD LANDING"
540 GOTO 110
550 END

```

not be a problem. The selection of speed, however, presents an opportunity that is unique to the user of a dedicated system. With a little trial and error, it should be possible to find the step size which causes your system to take exactly 1 second to calculate and display the speed and position 1 second into the future. One machine might do 100 steps of 0.01 seconds and then print the speed, etc. Another might do 64 steps of 0.015625 seconds before displaying new results. Still another, with slow peripherals, might output speed and position only once every 2 or 3 seconds. In any case, as long as the simulated data appears at

the same time that real data would, your system will be said to be running a real time simulation. A real time lunar lander game gives you exactly the same time to react as would be given a real excursion module pilot.

To help you implement this idea on your own system, a BASIC language program has been included with this article as listing 1. It should be easy to follow, but a few points are worth explaining. At each step the program will need to obtain the thrust settings. This is done through a function called `USR`. Because systems differ widely, the content of `USR` is left to you. Some systems will be able to use a register to hold the thrust; others will access memory location; and some may have to query an input port. Also left to the user is the manner in which the thrust settings are updated. Obviously, they cannot be entered at the keyboard for each 0.01 second step. The keyboard could be used via an interrupt routine however. Ideally, you could implement Thomas Buschbach's joystick interface (March 1977

BYTE, page 88) to allow continuous control of thrust in both degrees of freedom. What began as a simple game will now have become a real time lunar landing simulator requiring quick thinking and a good bit of practice to master.

If you use this idea then my article will have succeeded in its purpose of introducing some of the basic concepts of simulation. Techniques like separating the problem into degrees of freedom, determining the effect of each force separately, and stepping the simulation into the future are all fundamental to any prediction of motion. The differences between this lunar lander game and the complex simulations used in the space program lie in the way forces are determined and in the numerical methods used to calculate speed and position. In succeeding articles, other applications for simulation on microcomputers will be discussed as a means for demonstrating some of those advanced techniques. For now, try applying the ideas presented here to create a game of your own. ■

Two readers of BYTE Magazine submitted recommendations for modifications to the Lunar Lander Algorithm. These were printed in the BYTE's Bugs section of the April 1978 issue.

One Step at a Time

I was glad to see S P Smith's article on simulation in November 1977 BYTE, page 18.¹ That happens to be my specialty and an area that I feel has been neglected in personal computing. I'm afraid Steve erred in his discussion of the numerical integration, though. Over a given step, the average speed is equal to half the sum of the initial and final speeds. In the example, the average speed is $1/2 (100 + 99.16) = 99.58$ meters per second. Thus the LEM will fall 9.958 meters in 0.1 seconds, and the new position will be 10000. $-9.958 = 9990.042$. Admittedly, the difference is small and if we take a small enough step size, it becomes negligible. However,

steps of integration cost computer time, and if we're going to have a real time simulation, we need to get the most accurate integration for a given step size that we can. Just for the record, there are many much more accurate algorithms for numerical integration. At work we use a second-order method for all real time work. It's hardly more complicated than Steve's (requiring saving only one back point) but about two orders of magnitude more accurate.

Dr Jack W Crenshaw
2114 Cecille Dr SW
Huntsville AL 35803

Improving an Improvement

In the November 1977 BYTE an article called "An Improved Lunar Lander Algorithm" by Stephen P Smith (page 18)¹ has one small bug in it. If the user is using an interactive terminal to input the values for vertical and horizontal thrust, a situation can arise where the LEM can gain mass. This is how: if the value for $USR(1.)$ is zero, simulating free fall, and the value for $USR(2.)$ is

negative; then the equation $M=M-(A+B)*I*D$ results in $M=M-(0+(-B))*I*D=M+B*I*D$. In order to avoid this situation, we suggest that your interactive users change the mass equation to $M=M-(A+ABS(B))*I*D$. This new equation also prevents the case where the mass remains the same after a burn.

R Jovonovich
G Leenarmen
Sperry Univac
2276 Highcrest Dr
Roseville NM 55113

¹ page 35 of this edition.

Simulation of Motion

Part 2: An Automobile Suspension

Stephen Smith

Have you ever taken your system out to a club meeting or demonstration, only to find that something is ruining your car's handling? Was it because of the heavy power supply in the back seat? Would heavy duty shock absorbers help? You can answer these questions using your personal computer and the simulation techniques found here.

Part 1 introduced some basic ideas used in simulating motion. A games application was used as an example. In this section I shall expand on that base, explain some additional ways that forces can act, and demonstrate a more accurate technique for computing speeds and positions. The example I'll use will be a simulation of an automobile suspension and its response to a varying road surface. Automobile enthusiasts will be able to see how different springs and shock absorbers would affect the way a car rides. More important, all computer users will acquire some additional tools to use in their own simulations and gain insights into new applications for their personal systems.

First, let's review the basic points made in the last article. When beginning a simulation, you will first divide the motion being simulated into degrees of freedom. In other words, you will decide which motions you want to simulate, up and down, side to side, etc. From then on, calculations will be made separately for each degree of freedom. Next you will decide which forces are acting in each direction and determine how much each force would change the speed of some object in 1 second. If you use the metric system of units, the change, or acceleration (in meters per second per second), will be exactly equal to the force (in newtons) divided by the mass of the object (in kilograms). You will now be ready to predict the speed and position of the object at a step of D seconds into the future. Add up the effects of the individual forces. Multiply the total by D (the step

size) and add the product to the present speed. This is the speed of the object at a time D seconds into the future. Now multiply the speed by D and add that product to the present position. This is the position the object will take in D seconds. The simulation program will now calculate new values for the forces and mass and step the simulation forward once more. The process will continue until an end condition is reached.

In the lunar lander game simulation, two degrees of freedom were considered, up and down, and side to side. The up and down or vertical motion was affected by gravity and thrust. The side to side or horizontal motion was affected only by thrust. Both of these forces were determined independent of the speed and position of the lander. Gravity provided a constant change in speed, and thrust was controlled by the user. In this article we will explore variable forces which are not determined by the user, but directly by the speed and position we are simulating.

As mentioned earlier, the example we'll use is an automobile suspension, the parts which connect the wheel to the body. The most important of these parts are the spring and the shock absorber. We will assume that there are other parts which keep the wheel from moving back and forth, but only the wheel's up and down motion will be considered (see figure 1). Of course, the entire car can also move along the road. We will consider that as a second degree of freedom. Let's examine separately the forces that contribute to vertical and horizontal motion.

Motion down the road results when the car's motor, through the wheels, pushes the car forward. Air resistance and rolling friction try to slow it down. To simplify the simulation, we will assume that these forces balance each other exactly. This means that the speed along the road will not change. If the speed starts at some

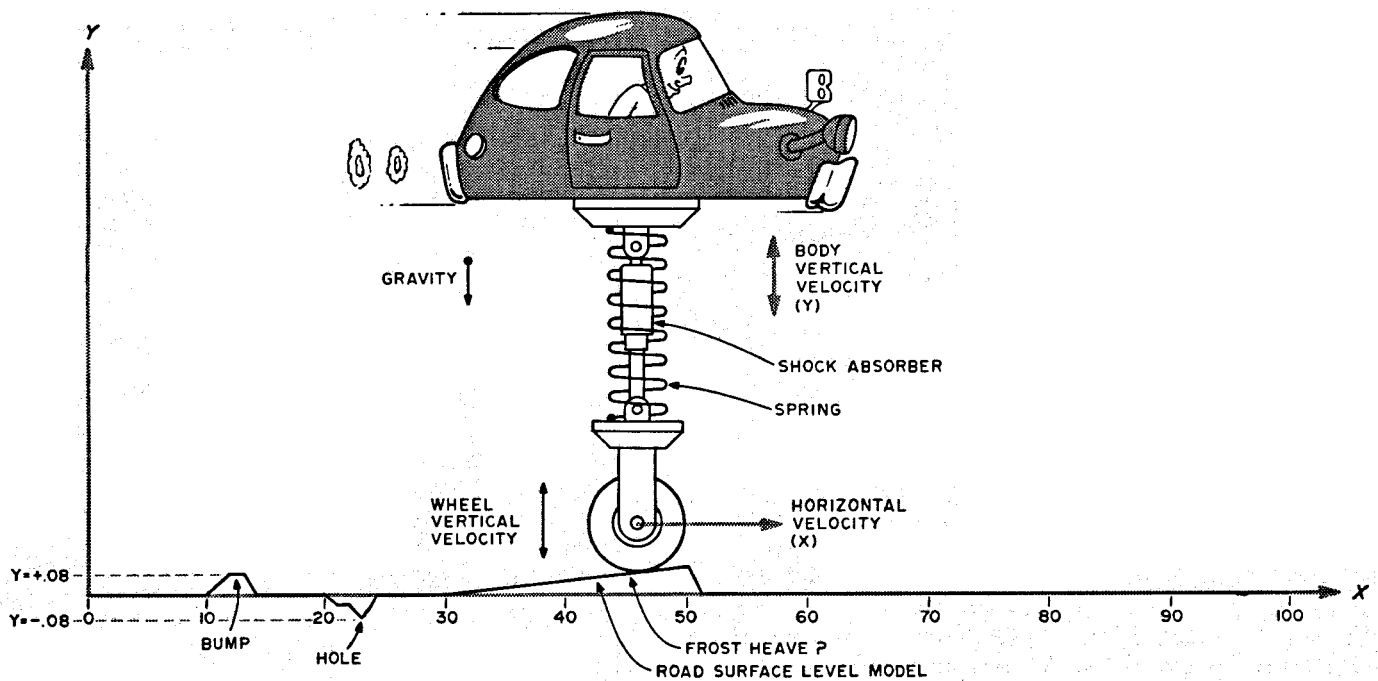


Figure 1: A conceptual model of the "automobile" (unicycle, rather) which is modelled in the sample program of listing 1 as discussed in this article. The wheel in this model tracks the road surface exactly, and has its own vertical velocity due to the horizontal velocity interacting with bumps in the road. The actions of the wheel in turn couple through the spring and shock absorber suspension to the "body" of the automobile. The purpose of this simple model is to calculate the vertical position of the car body at any given point down the road, given the effects of gravity, shock absorber, spring and excitement provided by the bumps and holes. The table in the figure is taken from lines 605 and 606 of the BASIC program of listing 1, and is used to plot the road surface. A better dynamic model of a car would have many more degrees of freedom than this simple model.

value other than zero, the horizontal position will change. As we will see later, the simulation program must keep track of the position along the road, because it will determine how the wheel, and in turn the body, moves up and down.

In the vertical degree of freedom we will need to consider gravity. You will remember from the last article that to simulate gravity a program subtracts a constant value from the speed for each unit step. (Speed and position are considered positive if they are directed upward.) On the earth the gravitational acceleration constant is 9.8 meters per second per second, so for each second of simulated time, velocity changes by 9.8 meters per second. Since the car obviously does not continue to move downward, there must be other forces balancing gravity. These are produced by the spring and shock absorber, and are determined by the vertical speed and position of the body.

Let's examine the spring first. At its normal length (often called the free length) a spring produces no force at all. If it is compressed, in other words forced to become shorter, it will push back on whatever is compressing it. The shorter the spring is forced to become, the harder it will push back. This is an example of a force that depends upon position. In the automobile

example, as gravity pulls the body down, the spring is compressed. The spring begins to push upward on the body, and at some point the two forces balance each other. The body will eventually come to rest there.

Knowing a little information about the spring we can compute that point. Springs produce forces which are equal to the distance they are compressed times a constant. The metric units for the constant are newtons per meter. Sample values are shown in table 1, column a. Suppose that gravity exerts a force of 5000 newtons on the car body; then a spring with a constant of 100000 newtons per meter would have to be compressed .05 meters ($100000 \times .05 = 5000$.) to balance the pull of gravity. At this point the system would be in equilibrium.

What about the shock absorber? It was designed to produce a force that depends not on how far it is compressed, but on how fast it is being compressed. The faster you try to move it, the harder it resists being moved. Like the spring, a constant is used to calculate the force, this time multiplying the speed. The metric units for this constant are newtons per meter per second and some representative values are shown in table 1, column b. At equilibrium there is no motion, so the shock absorber produces no force. If you were to push down the car

body at a speed of 2 meters per second, a shock with a constant of 50 would resist that motion with a force of 100 newtons (50×2). When you let up on the body, the spring would exert a greater force than gravity and the body would move upward. The shock absorber would also resist that motion. This action is called damping. The damping in an automobile suspension must be carefully chosen so that the body returns quickly to equilibrium, but does not continue to bounce back and forth for very long afterward.

Armed with your present knowledge of simulation you should be ready to make just such a choice using a trial and error approach. Calculate the forces on the body, and then use them to find the speed and position one step into the future. That speed and position will be used to calculate new values for the forces, which in turn will be used to step the simulation forward once more. Repeating the process continuously, you will simulate the motion of the car body. Try different values for the spring and damping constants until the desired output is achieved for a given set of inputs.

The inputs, you'll remember, are going to be determined by the simulated position in the horizontal degree of freedom. At each position along the road the input routine will determine the height of the road surface above or below normal. If we assume that the wheel does not leave the road this will also give us the up and down motion of the wheel. The data can be stored in a table in memory. By entering different values for the horizontal speed at the start of the simulation, we can also vary how fast the car will pass over our model road. At each step the program will enter the table to find the road height which corresponds to the current horizontal position.

This method will work as long as there is an entry in the table for every horizontal position we will find. That could be a very big table, especially if the step size is small. To eliminate the need for large tables, we can use a technique called interpolation. Very simply, interpolation is done like this. When the program enters the table, but doesn't find an entry exactly equal to the current horizontal, it finds the next smaller entry and the next larger entry. An interpolation formula is then used to figure out where the present position falls between the two table entries, and to calculate the road surface which lies at the same point between the corresponding table entries of road height. For example, suppose a program entered table 2 to find the road surface corresponding to a horizontal position of 11. It would find entries at

Vehicle	(a)	(b)
	Spring	Damping
Full size car (LTD, etc)	3200	1450
Intermediate (Torino, Cutlass, etc)	3000	1200
Compact (Nova, Aspen, etc)	2800	1000
Subcompact (Vega, Pinto, etc)	2600	700
Add 20% for heavy duty suspension.		
Subtract 20% for front wheel.		

10 and 12 with corresponding road heights of 0.0 and 0.08. Because 11 lies halfway between 10 and 12, the interpolation formula will find a corresponding road surface that lies half way between 0.0 and 0.08 or 0.04. There are other interpolation formulas that use three, four, or more of the table points, but this method using two is generally accurate enough with a reasonably detailed table. To simplify your implementation of interpolation, I have included a BASIC function in the program of listing 1 which uses the 2 point method. Users can simply place their own tables in the data statement and use the function in their programs, or they can follow through the equations and implement them directly.

In our automotive simulation, the interpolated table data will give us the vertical road and wheel position. The difference between this and the vertical position of the body will be the amount the spring is compressed. We can quickly calculate the resulting force. If the simulation program retains the wheel's position from the previous step, it can also calculate the wheel's vertical speed. Reversing the equation used to find a new position, the speed is equal to the difference in the two positions divided by the step size. If the wheel moved from .08 meters to .04 meters in a step of 0.01 seconds, its speed would be $(.04-.08)/.01 = -4.0$ meters per second. The difference between this speed and the speed of the body is used to calculate the force produced by the shock absorber. All these calculations are included in the BASIC program of listing 1. Readers who want more detail on the equations will find them there as well-commented program statements.

Also in that program is a new method for computing speeds and positions. The equations used in the lunar landing game worked fairly well when the forces did not depend upon the speed and position. In this simulation they do, and even small errors can snowball if not corrected. To do this, we will use a powerful numerical technique, one which uses the results from three previous steps to help predict the next, and

Table 1: Representative spring and damping constants for automobiles. The units are metric: the spring constant is quoted as newtons per meter of compression; the damping constant is expressed as newtons per meter per second.

Table 2: A sample road surface table. This table is used to draw the surface curve shown in figure 1.

Horizontal Position	Road Surface
0	0.0
10	0.0
12	0.08
13	0.08
14	0.0
20	0.0
21	-.04
22	-.04
23	-.08
24	0.0
30	0.0
50	0.1
51	0.0
100	0.0

which then goes back and corrects the step when the predicted results are available. It is called, logically enough, a predictor-corrector method. Rather than attempt to explain it here, I'll provide a BASIC programming example which you can adapt to your own simulations. Readers with a good background in math may wish to reference a book on numerical methods for more details. In either case you will have acquired a tool which will be very useful in future simulations.

Looking back over the two articles you should begin to see some ideas for your own simulations. They could involve

forces which are constant, user controlled, or which depend directly on the motion you are simulating. Inputs can come from your keyboard, from an analog device such as a joystick, or from tables interpolated by your program. The outputs might tell you how well you are playing a game, or which of several configurations is best for a design you are contemplating. In the next article I'll continue to expand on the types of forces considered. In particular, I'll show how you can handle forces which act in more than one degree of freedom and suggest some ways to handle rotary motion.■

```

100 REM SET PROGRAM CONSTANTS
110 K1=-100000
120 K2=-2000
130 M=500
140 U=10
150 D=.05
151 REM SET INITIAL SPEED AND POSITION
160 T=0
170 P1=9.8*M/K1
171 S1=0
172 REM FIND INITIAL ROAD SURFACE
175 I1=0
176 X1=0
177 Y1=0
178 X=0
182 GOSUB 600
183 I2=(Y-I1)/D
184 I1=Y
185 REM CALCULATE INITIAL FORCES
186 F1=(P1-I1)*K1/M
187 F2=(S1-I2)*K2/M
188 A1=F1+F2-9.8
189 REM SET PAST DATA EQUAL TO INITIAL DATA
190 S2=S1
200 S3=S1
210 S4=S1
230 A2=A1
240 A3=A1
250 A4=A1

259 REM BEGIN SIMULATION
260 REM PREDICT SPEED AND POSITION
270 S=S1+D/24*(55*A1-59*A2+37*A3-9*A4)
280 P=P1+D/24*(55*S1-59*S2+37*S3-9*S4)
281 X=X+U*D
282 REM FIND NEW ROAD SURFACE
283 GOSUB 600
284 I2=(Y-I1)/D
285 I1=Y
290 REM PREDICT NEW FORCES
291 F1=(P-I1)*K1/M
292 F2=(S-I2)*K2/M
300 A=F1+F2-9.8
310 REM CORRECT SPEED AND POSITION
320 S=S1+D/24*(9*A+19*A1-5*A2+A3)
330 P=P1+D/24*(9*S+19*S1-5*S2+S3)
340 REM CORRECT FORCES AND UPDATE SAVED DATA
341 F1=(P-I1)*K1/M
342 F2=(P-I2)*K2/M
350 A4=A3
360 A3=A2
370 A2=A1
380 A1=F1+F2-9.8
390 S4=S3
400 S3=S2
410 S2=S1
420 S1=S
430 P1=P
440 T=T+D
450 PRINT T,S1,P1
460 IF X<100 THEN 270
470 END

600 REM INTERPOLATE TABLE TO FIND
601 REM VALUE OF Y CORRESPONDING
602 REM TO GIVEN VALUE OF X
605 DATA 0,0,10,0,12,0.08,13,0.08,14,0.20,0.21,-0.04
606 DATA 22,-0.04,23,-0.08,24,0.30,0.50,0.1,51,0.100,0
611 REM TABLE FORMAT IS X(1),Y(1),X(2),Y(2), ...
620 IF X<X1 THEN 670
630 X2=X1
640 Y2=Y1
650 READ X1,Y1
660 GO TO 620
670 Y=Y2+(Y1-Y2)*(X-X2)/(X1-X2)
680 RETURN

```

Automobile Suspension Simulator

Listing 1: This program was written to help interested readers follow the mathematics of the accompanying article. Particular attention should be paid to the interpolation subroutine and to the equations for the predictor-corrector method of predicting future positions and velocities. The program was not intended to be efficient; readers will surely be able to shorten it once the method is understood. The following table defines the variable names I've used.

K1 = spring constant

K2 = damping constant

M = mass supported by the spring

V = horizontal speed of the entire car

D = time step size

T = elapsed time in the simulation

P, S, A = predicted values for vertical position, speed, and total effect of forces

P1, S1, A1 = present values of vertical position, speed, and total effect of forces

S2, A2 = speed and effect of forces one step past

S3, A3 = speed and effect of forces two steps past

S4, A4 = speed and effect of forces three steps past

F1 = change in speed due to spring

F2 = change in speed due to damping (shock absorber)

I1 = current vertical position of the wheel

I2 = current vertical speed of the wheel

X = current position of car along the road

Y = road height at position X

X1, Y1, X2, Y2 = table entries for positions immediately greater than and immediately less than the current value of X

I expect it will occur to many of you that graphic rather than printed output will make this program much clearer. The waveform produced by a plot of the data would give you a much better feel for the motion of the car body. For the example in this listing, try plotting position from -0.1 meter to +0.1 meter versus time from 0 to 100 seconds.

One final note: to avoid losing data, it is important that the interval between points of the table in the interpolation subroutine is larger than the distance the car moves in one step. In other words, if you want to model a road that changes rapidly, you will have to reduce the step size (D) to a value less than the minimum of $(X(n) - X(n+1))/V$.

Simulation of Motion

Part 3: Model Rockets and Other Flying Objects

Stephen Smith

Since becoming involved in personal computing, I've only met a few real applications oriented users. Most microcomputer owners are either hardware oriented, eg: hams or other electronics hobbyists, or they are software hackers. Both groups tend to love their machines for their own sake and not necessarily because they are useful. The users I've met who are more interested in the answers they get than in how they got them have all been running financial programs. Despite this thin showing, I believe that the next large group to "discover" personal computing is going to be applications oriented. They will be the business people and hobbyists who need more computing power than is available in a pocket calculator, but who can't justify access to a large computer.

Among this group will be the model rocket and aircraft builders. Those people delight in creating miniature NASAs, but they have always lacked one important resource, computing power. The government and the aerospace industry invest a great deal of effort in simulating flights long before any hardware is put together. Few hobbyists could do this until now. In this article, I'll show how a microcomputer can be used to simulate the flights of model or amateur rockets and aircraft. The simulations can be used as aids in design and "mission" planning. I also hope to demonstrate the desirability of personal computing as an adjunct to other pastimes.

As in my previous articles, these simulations are intended to serve as examples. The techniques involved can be applied in almost any real world application. The lunar lander game, for example, served to illustrate how

simulations are separated into degrees of freedom, and how speed and position are predicted for steps into the future. Those concepts were also applied when we simulated the motion of an automobile suspension system. That simulation illustrated how forces could depend directly on the motion being simulated. It also served to introduce some powerful numerical techniques. All of these concepts will apply to the flight simulations. In addition, I'll show how to calculate the effects of a force which acts in two degrees of freedom at the same time. I'll also introduce angular motion and demonstrate the way in which a simulation keeps track of how fast a body is turning and where it is pointing. While developing a flight simulation in detail, I'll try to point out specific areas where these new techniques can be applied to the earlier applications.

Body	C_d
flat plate (1 square meter)	0.7
sphere (0.1 meter diameter)	0.003
airplane body (2 square meters)	0.08
wing or fin edgewise (1 square meter)	0.012
model rocket (2 cm diameter)	0.0001
automobile (2 square meters)	0.6
motorcycle and rider (2/3 square meters)	0.25

Table 1: Drag coefficients for various bodies. These coefficients include the body area and air density term ($1/2 \times 1.192 \text{ kg/m}^3$). They are intended to be used in an equation of the form:

$$\text{DRAG} = \text{SPEED}^2 \times C_d$$

If larger or smaller bodies are used, a simple ratio of areas will convert the coefficient.

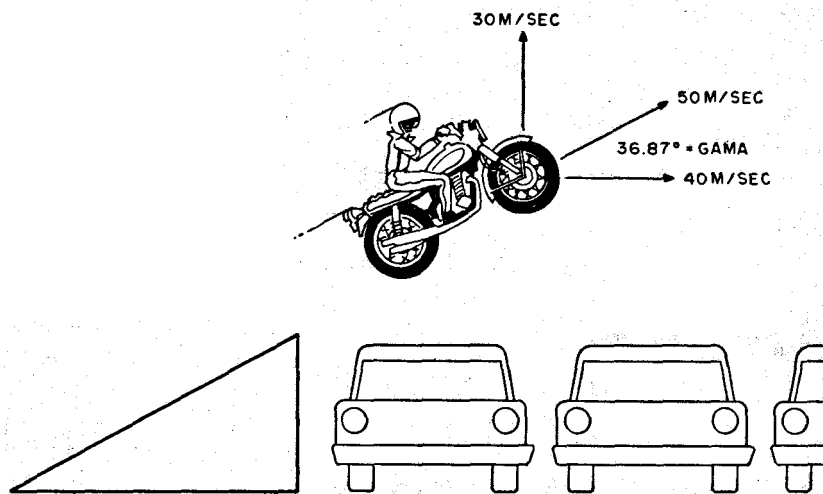


Figure 1: The total speed and flight elevation angle can be calculated from the horizontal and vertical speed components.

Let's begin, in fact, by outlining a game simulation you can program. In the lunar lander game, aerodynamic forces were neglected, but these forces must be considered for atmospheric flight. They also present a good example of forces which act in more than one degree of freedom. We will investigate them through the use of a simple game I'll call EVEL. The object of the game is to select the ramp angle and speed with which to leap a motorcycle over a given number of cards and land successfully on the downward ramp. The motion will be in two degrees of freedom, vertical and horizontal. The forces will be gravity and aerodynamic drag.

We have seen in the previous articles how gravity affects speed, and most people have an intuitive understanding of drag. Drag is the force you feel when you hold your hand out the window of a moving car. It is the resistance of air to a body moving through it and it acts directly opposite the motion. Drag is calculated in much the same way as the force created by an automobile shock absorber. In that example the force was equal to the speed multiplied by a damping coefficient. To calculate drag, we will multiply the speed squared by a constant called the drag coefficient (symbolically C_d). Drag coefficients for some common bodies are given in table 1. C_d takes into account the size, shape and surface texture of the body. In our simulations, it will also include a factor for the density of the air (1.19 kg per cubic meter). More detailed simulations will take into account the changes in air density and temperature which occur at higher altitudes and adjust the aerodynamic forces accordingly. To avoid this complication, we will restrict our simulations to altitudes within a few thousand meters of

sea level. The formula we will use for calculating drag is $DRAG = SPEED^2 \times C_d$.

If the only motion is upward, the drag acts only in the vertical degree of freedom. There are also cases in which it acts only horizontally; but in general, there will be motion in both directions, and the drag, which acts directly opposite the motion, will be felt in both degrees of freedom. Because of its dependence on the square of the speed, we cannot calculate separate vertical and horizontal drags. We must calculate one force and apportion it between the two degrees of freedom.

Figure 1 shows a typical case. Here a daredevil motorcyclist has just left his takeoff ramp. Suppose we know from a previous simulation step that his vertical speed is 30 meters per second (m/sec) and his horizontal speed is 40 m/sec. To calculate drag, we must first find the total speed. Our fortunate selection of degrees of freedom now becomes apparent, because the vertical and horizontal velocities can be seen to form two sides of a right triangle. We can compute the third side, or hypotenuse, by applying the theorem of Pythagoras ($C^2 = A^2 + B^2$). The total velocity will be equal to the square root of the sum of the squares of the speeds in each degree of freedom. In this case, the daredevil is moving at $\sqrt{30^2 + 40^2} = 50$ m/sec. Using C_d from table 1, we calculate a drag of $0.25 \times 50^2 = 625$ newtons, acting at some angle between horizontal and vertical. This angle is called the flight elevation (symbolically GAMA in some computer programs). It can be found using a little trigonometry. If we let horizontal be 0 degrees, and let vertical be 90 degrees, then GAMA is equal to the arc tangent of the vertical divided by horizontal velocity. In this case, $GAMA = \arctan(30/40) = 36.87$ degrees. Knowing the angle, it is easy to apportion the drag. The forces which result are called components of drag. The vertical component (symbolically D_v is given by $D_v = DRAG \times \sin(GAMA)$). The horizontal component, D_h , is given by $D_h = DRAG \times \cos(GAMA)$. In the current example, $D_v = 625 \times \sin(36.87) = 375$ newtons, and $D_h = 625 \times \cos(36.87) = 500$ newtons. You can check these calculations by noting that $\sqrt{375^2 + 500^2} = 625$. Readers familiar with trigonometry will be able to confirm that

$$DRAG = \sqrt{D_v^2 + D_h^2}$$

in every case. This is the same formula we used to find the total speed, so it should not be surprising to find that the vertical and horizontal speeds are also referred to as components.

The effect that the components of drag have on the components of speed depends, of course, on the mass. If our daredevil is fairly small, and rides a light motorcycle, the total mass in flight might be 150 kg. During a step of 0.1 seconds, the horizontal speed will decrease by $500/150 \times 0.1 = 0.333$ m/sec. A similar change occurs in the vertical speed, but there we must also include gravity. Remember from the previous simulations that each second, gravity subtracts 9.8 m/sec from the vertical speed. In 0.1 seconds it will change from 30 m/sec to $30 - (375/150 + 9.8) \times 0.1 = 28.77$ m/sec. Knowing the new speeds, you can compute the new position, the new drag and the new flight elevation. The simulation can be stepped forward again and again until the daredevil returns to earth.

Because the drag components depend on the square of the speed, it will probably be necessary to use the predictor-corrector formulas from my previous article to obtain realistic results. The initial conditions of total speed and ramp angle must be chosen. For the first simulation step set GAMA equal to the ramp angle and let the vertical speed component equal $\text{SPEED} \times \sin(\text{GAMA})$ and the horizontal component equal $\text{SPEED} \times \cos(\text{GAMA})$. Figure that a car is about 6 meters long, a bus about 15 meters, and the Snake River Canyon is 1451 meters wide. Good luck.

In many respects, a rocket or aircraft in flight is much like our daredevil. It will be moving horizontally and vertically and will be acted upon by gravity and drag. There are some other forces to be considered, however. For example, early in a rocket's flight, its engine will be producing thrust. Unless the rocket is pointing directly upward or directly parallel to the ground, we will also have to apportion this force between horizontal and vertical directions.

One way to do this is to assume that the rocket always points in exactly the direction it is moving. The flight elevation angle, GAMA, can then be used to apportion thrust just as it was used for drag. At each simulation step, the program can interpolate a table to find the value of thrust corresponding to the current time. It then computes the components and applies them to predict new speeds and position. In this manner we can build a two degree of freedom rocket trajectory simulation. This simulation might also be used for a game. Set up a duel between two artillery battalions, or better still, program a real time simulation like the lunar lander game of my first article and try to hit a moving target.

For most flight vehicles, the tendency to point in the direction of movement is a desirable characteristic. Unfortunately, this is not generally the case. Vehicle imperfec-

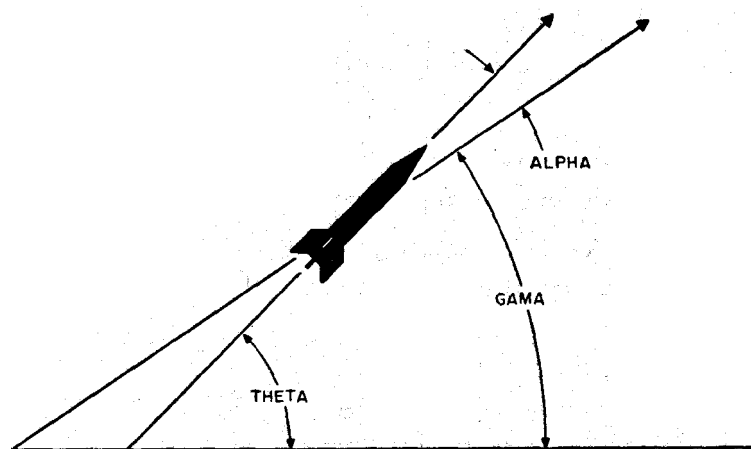


Figure 2: The direction of flight is called the flight elevation (GAMA). The direction the rocket is pointing is called the body elevation (THETA). The difference between them is called the angle of attack ($\text{ALPHA} = \text{THETA} - \text{GAMA}$).

tions, the effect of wind, and just the fact that it takes a finite amount of time to turn the vehicle, all affect the pointing of a rocket. In order to indicate where the rocket is pointing, we define another angle, the body elevation angle (symbolically THETA). THETA tells us where the vehicle is pointed and is used to apportion thrust. The difference between THETA and GAMA is called the angle of attack (symbolically ALPHA). It is illustrated in figure 2.

Unlike GAMA, there are no components which may be used to compute THETA. We must keep track of it with a third degree of freedom. Just as altitude is calculated as the position in the vertical degree of freedom, THETA may be calculated as the position in this third, or angular, degree of freedom. Angular motion is handled in exactly the same way as the linear motions we have already been simulating. Just as position has its angular equivalent, so do speed, force and mass. It is as easy to calculate the speed with which a body turns as it is to find the speed with which it falls.

First we must calculate the angular equivalent of a force. These are called moments (also called "torque"), and have metric units of newton meters. As the units imply, each moment has a force as part of its definition, but it also depends on how much leverage the force has. For example, suppose you apply a force of 10 newtons to the end of a 15 cm (0.15 meter) wrench. A moment of $10 \times 0.15 = 1.5$ newton meters will be transferred to the bolt. If a 25 cm wrench were used, a moment of $10 \times 0.25 = 2.5$ newton meters would result, and the bolt would be proportionately tighter. In rocket flights a moment results as an aerodynamic effect whenever THETA is not

```

005 REM PITCH PLANE TRAJECTORY SIMULATION
010 REM INITIALIZE VARIABLES
020 DATA 0,0,0,0,0,0,0,0,0,0,0,0
030 READ U,M,U1,M1,U2,X1,Z1,Q1,F1,M1,I1,K
040 REM INITIALIZE TIME(T),STEP SIZE(H), PRINT INTERVAL(KI)
050 H=0.01
060 LET T=0
070 LET T1=T
080 LET KI=0.1/H
090 REM SET INITIAL ALTITUDE (Z1>0)
100 LET Z1=1
110 REM SET DRAG(D) AND MOMENT(P) COEFFICIENTS
120 LET D=-1.0E-4
130 LET P=-3.0E-5
140 REM SET LAUNCHER LENGTH(R) AND ELEVATION(L)
150 L=90
160 PRINT "LAUNCH ELEVATION =" ;L;" DEGREES"
170 LET L=L+0.01745
180 LET L1=L
190 LET G=L
200 LET R=Z1+1
210 REM SET WIND SPEED
220 N=5
230 PRINT "WIND SPEED =" ;N;" METERS/SECOND"
240 LET U1=N
250 REM END INITIALIZATION, BEGIN TRAJECTORY
260 PRINT "TIME          ALTITUDE      RANGE      SPEED"
270 PRINT "(SEC)           (METERS)    (METERS)   (M/SEC)"
280 REM GET THRUST(F), MASS(M), & MOMENT OF INERTIA (I)"
290 GOSUB 950
300 LET T=T+H
400 LET K=K+1
410 REM PREDICTOR FORMULAS
420 REM B,M,Z,ARE VERTICAL ACCELERATION,SPEED & POSITION
430 LET B1=(SIN(G)*D*U2+SIN(L)*F)/M-9.8
440 LET M=M1+H*B1
450 LET Z=Z1+H*M1
460 REM A,U,X ARE HORIZONTAL ACCELERATION, SPEED & POSITION
470 LET A1=(COS(G)*D*U2+COS(L)*F)/M
480 LET U=U1+H*A1
490 LET X=X1+H*(U1-N)
500 LET U2=(U+N)^2+M^2
510 LET U=SQR(U2)
511 REM NO ANGULAR MOTION ON LAUNCHER
512 IF Z<R THEN 590
513 LET G=ATN(M/(U+N))
514 IF U+N>0 THEN 530
515 LET G=G+3.14159
520 REM NEGLECT ANGULAR MOTION AFTER BURNOUT
530 IF F=0 THEN 610
535 REM C,Q,L ARE ANGULAR ACCELERATION, SPEED & POSITION
540 LET C1=P*(L-G)*U2/I
550 LET Q=Q1+H*C1
560 LET L=L1+H*Q1
590 GOSUB 950
600 REM CORRECTOR FORMULAS
610 LET B=(SIN(G)*D*U2+SIN(L)*F)/M-9.8
620 LET M=M1+H/2*(B+B1)
630 LET Z=Z1+H/2*(M+M1)
640 LET M1=M
660 Z1=Z
680 LET A=(COS(G)*D*U2+COS(L)*F)/M
690 LET U=U1+H/2*(A+A1)
700 LET X=X1+H/2*(U+U1)
710 LET U1=U
720 LET X1=X
730 LET U2=(U+N)^2+M^2
740 LET U=SQR(U2)
750 REM NO ANGULAR MOTION ON LAUNCHER
751 IF Z<R THEN 880
752 G=ATN(M/(U+N))
753 IF U+N>0 THEN 770
754 G=G+3.14159
760 REM NEGLECT ANGULAR MOTION AFTER BURNOUT
770 IF F=0 THEN 880
780 LET C=P*(L-G)*U2/I
790 LET Q=Q1+H/2*(C+C1)
800 LET L=L1+H/2*(Q+Q1)
810 LET Q1=Q
820 LET L1=L
880 IF K<KI THEN 390
890 PRINT T,Z,X,U
900 LET K=0
920 IF Z>0 THEN 390
930 REM STOP WHEN ALTITUDE RETURNS TO ZERO
940 STOP
950 REM INTERPOLATE TIME TABLE FOR F,M,& I
960 DATA 0,5.6,0.0458,1.1E-4,0.1,13.3,0.0446,1.1E-4,0.2,5.6,0.0434,1.1E-4
970 DATA 1.6,5.6,0.0337,1.0E-4,1.7,0.0,0.0333,1.0E-4,1.00,0.0,0.0333,1.0E-4
980 IF T<T1 THEN 1050
990 LET T2=T1
1000 LET F2=F1
1010 LET M2=M1
1020 LET I2=I1
1030 READ T1,F1,M1,I1
1040 GO TO 980
1050 LET H4=(T-T2)/(T1-T2)
1060 LET F=F2+(F1-F2)*H4
1070 LET M=M2+(M1-M2)*H4
1080 LET I=I2+(I1-I2)*H4
1090 RETURN
1100 END

```

Listing 1: This BASIC program illustrates the use of an angular degree of freedom to apportion a force between two linear degrees. It simulates the flight of a model rocket that might be built from widely available kits. During the early part of the flight (approximately 1 meter), the rocket rides on a guiding rod. The rod prevents the body from turning, so angular motion must be suppressed in the simulation. After leaving the launcher, the vertical, horizontal and angular pitching motions are all considered. A short time later, however, the fuel will be exhausted and the thrust will drop to zero. Because the only reason for computing THETA was to apportion the thrust, angular motion may be neglected from this point until the vertical position becomes zero (rocket returns to earth) and the simulation ends.

Because the forces and moments in this simulation depend on the motion, the predictor-corrector method has been applied. To conserve space, however, it is not the fourth order method used in the automotive simulation of Part 2 of this series of articles, but a shorter second order version. Improved accuracy would result if you were to implement the longer formulas. You may also want to customize the output. For example, a plot of angle of attack versus time might be interesting. To print out ALPHA in degrees, use $(L-G) \times 57.295$ to convert from units of radians. Special output exactly at apogee (maximum altitude), impact, or motor burnout might also be helpful in evaluating performance. The code is well-commented, so feel free to dig in and adapt this program to your own needs.

Figure 3: The program of listing 1 can be used to determine the best launcher elevation for any given wind.

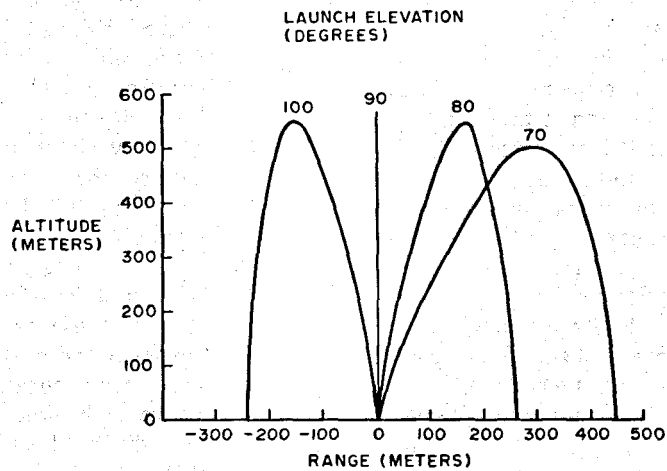


Figure 3a: Trajectories with no wind.

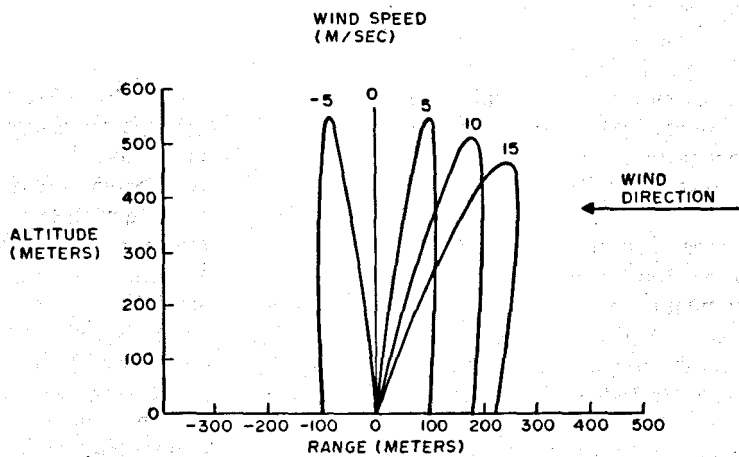


Figure 3b: 90° launch with wind.

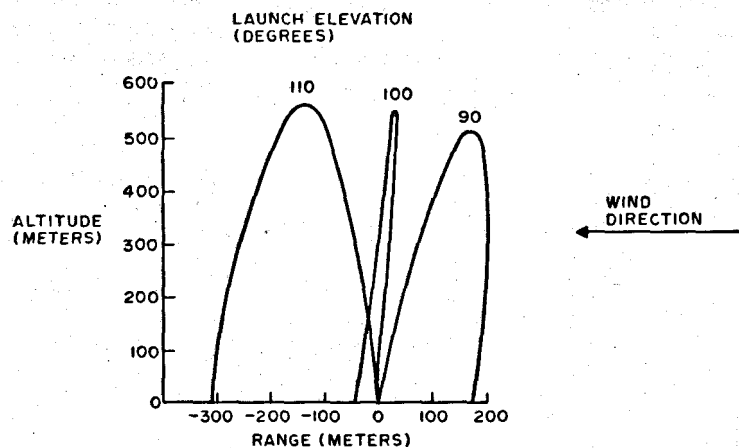


Figure 3c: Trajectories with 10 m/sec wind.

equal to GAMA. It has its own coefficient, C_m . For small angles of attack, this coefficient is multiplied first by the square of the speed (linear speed, not angular) and then by the angle of attack ($\text{ALPHA}=\text{THETA}-\text{GAMA}$). The aerodynamic moment is therefore caused because the rocket is trying to point itself in the direction it is moving.

The next element to be considered in the angular equation is the equivalent of mass. This is called moment of inertia, and the leverage concept applies to it also. The moment of inertia (usually shown symbolically as an upper case I) depends on the mass of the body, and also how widely spaced the mass is. For example, a 50 kg set of bar bells would have a much larger moment of inertia than a single 50 kg weight. The units of I are kilogram meters. Like mass, moment of inertia is a property of matter. In our rocket simulation, both may change as fuel is burned, but their values can be determined at any given time. It is general practice to construct tables of mass and moment of inertia which parallel the thrust table we have already introduced. In each simulation step, our program will interpolate the table to find the current value of moment of inertia and use it to find the effect of the moments. The value of the metric units will now become apparent.

Just as the force in newtons was divided by the mass in kilograms to find exactly the change in speed each second, so the moment in newton meters can be divided by the moment of inertia in kilogram meters to find exactly the change in angular speed. In angular degrees of freedom the units of speed are radians per second and the position will be computed in radians. There are 2π radians in a full circle, so 1 radian equals 57.296 degrees. Because most BASIC interpreters perform trigonometric calculations in radians, these units are to be preferred. Conversion to degrees is easily made for input and output.

Once the effect of each moment is known, it is multiplied by the time step size and

added to the speed in exactly the same technique used for linear motion. Similarly the angular speed is multiplied by step size to update the angular position. The predictor-corrector method may be applied by saving moments, and speeds from previous steps. I have included a BASIC program with this article which involves both the predictor-corrector formula and angular motion in a three degree of freedom (pitch plane) trajectory simulation. Noting the similarity between the angular and linear equations used there should make this technique easier to understand.

The concept of angular motion is easily transferred to other applications, but to make best use of it you really need some familiarity with the forces and moments which may appear. For example, an automotive enthusiast will already understand how the road surface induces motion in the body of the car. We saw previously how to simulate that motion for one wheel. If you include two wheels, the front and back on one side, a second degree of freedom in body motion is introduced. Most people would recognize that, but only someone familiar with automobiles might realize immediately that the two ends of the body will not move up and down entirely independently. The second degree of freedom must be an angular one. It will measure the pitching motion of the car while the original degree measures its overall up and down motion. The forces created by each set of suspension parts will contribute to the linear motion, and will be multiplied by their leverage (perhaps their distance from the driver's seat) to determine their affect on the angular motion. At each step, the angular motion will be combined with the linear motion to compute new forces, moments, etc, and the procedure will begin again. Just how those angular motions are combined with the linear ones, and how the leverage of a force is determined, will be the subjects of the fourth and last article in this series.■

Simulation of Motion

Part 4: Extended Objects, Applications for Boating

Stephen Smith

Have you ever wondered why the shapes of boat hulls differ so widely? Boating enthusiasts know that certain designs will be best in lakes and rivers, and certain others in open seas. Some boats are much roomier than others; some are safer in rough water; but what penalties in stability and riding comfort might you pay for the extra room or seaworthiness? The motion of a boat depends on its response to the variety of waves it encounters. These motions can be simulated on your personal computer. You can determine how a given design will respond to any sea condition. The basic equations for stepping speed and position into the future will still apply, as they were discussed in the earlier articles of this series; but you'll also need some new techniques. As you implement this simulation, you'll discover that forces in a linear degree of freedom can also produce moments and their resulting motion in an angular degree of freedom. In this article, I'll show how that interaction is handled. I'll also introduce the concept of distributed forces, and a numerical technique to handle them. Although developed for a boating application, these new ways of calculating forces should find use in updating many of our previous simulations.

We have already seen quite a variety of ways to calculate forces. Gravity, a force present in every simulation in the last three articles, simply made a constant change in the vertical speed at each step. Thrust, used in rocket and aircraft simulations, came either from a user input or from a table interpolation. Forces in an automobile

suspension were found to depend directly on the vertical position (spring force) and the vertical speed (damping force). Aerodynamic forces were computed by multiplying a coefficient (ie: constant) by the sum of the squares of the speeds in each linear degree of freedom. While these examples cover most of the situations you are likely to encounter in simple models, any new simulation might present some unique requirement.

For example, in all the calculations, the forces have had one thing in common. They acted at a single point. We call such forces discrete. In reality, some are not discrete, but act at many points on a body simultaneously. These are referred to as distributed forces. Aerodynamic drag is a typical distributed force. Although we used the drag coefficient to calculate a single force, the retarding action of the air acts all over the body. A coefficient is just one tool used to convert distributed forces into discrete ones. Not all distributed forces can be converted using coefficients, so I'll introduce a more general technique using the boating simulation as an example.

The principle forces on a boat are gravity and buoyancy, the floating power of the hull. Because buoyancy is an upward push provided by the water, it is not difficult to see that it is a distributed force covering the entire area of the boat below the water line. Converting this distributed force to a discrete one will allow us to simulate the vertical motion of the boat.

Perhaps more important to the boat designer or buyer will be the angular, rolling

Wind Speed	Rivers	Lakes	Inlets	Bays	Open Sea
2 m/sec					
period (sec)	0.6	1.0	1.5	2.0	3.5
length (m)	0.56	1.5	2.3	3.1	20.0
height (m)	0.02	0.06	0.12	0.15	0.5
5 m/sec					
period (sec)	0.8	1.2	2.0	2.4	4.5
length (m)	0.1	2.25	5.0	9.0	30.0
height (m)	0.05	0.08	0.2	0.25	0.75
10 m/sec					
period (sec)	1.25	2.0	3.0	4.25	7.0
length (m)	2.4	6.25	14.0	28.0	80.0
height (m)	0.08	0.15	0.35	0.7	2.0
20 m/sec					
period (sec)	2.5	4.0	6.0	8.5	14.0
length (m)	10.0	25.0	56.0	110.0	300.0
height (m)	0.25	0.65	1.4	2.8	7.5

Table 1: Characteristics of waves. The height, period and length of waves all vary, but for certain conditions, average values have been established. The wave length and period are affected by the depth of the water. The height depends on the wind speed, how long it has been blowing, and the width of the body of water. Readers who want to model real sea conditions should find a good oceanography text, but the above summary should prove adequate for casual use.

and pitching motion of the boat. Angular motion was introduced in a rocket flight simulation (see page 45). In that case, it was entirely independent of the linear motion. At the end of the same article I suggested that the motion of an automobile body should also be simulated using an angular degree of freedom, but that the angular and linear motions could no longer be considered separately. This is also true in the boating example. The moments used to compute the angular motions will be calculated directly from the linear motions. Because the forces in the automotive example are discrete, we'll develop the technique to handle combined angular and linear motion using the distributed forces of the boat example. In that way, one simulation will serve to demonstrate both of the new concepts. I'll leave the development of a two or four wheel automobile suspension simulation to interested readers.

The motion of a boat is similar in many ways to that of the automobile body. When it is launched, a boat settles into the water in response to gravity. As the hull displaces more water, the buoyant force becomes larger, until at some point, it balances gravity and the boat stops sinking. This point is called equilibrium and is analogous to the equilibrium of an automobile suspension. Unless there is a disturbance, the boat will remain at equilibrium. In the automotive example, disturbances came in the form of a rising or falling road. With boats, we en-

counter a rising and falling sea, in other words, waves.

Sea waves occur in a variety of shapes. Their length (distance peak to peak), their height (distance peak to trough), and their period (time to rise and fall), all vary apparently at random. In fact, these parameters have fairly well defined relationships. Readers with an interest in modeling sea states should refer to a good marine science text. For this simulation, we'll represent waves with a sine function, and use the data in table 1 to compute their size.

Dealing just with forces for a moment, let's see how a small object is affected by wave motions. Figure 1 shows a bottle, floating in a body of water. We know from our previous simulations that every second, gravity subtracts 9.8 meters per second from the bottle's vertical speed. If the bottle is to remain stationary, the effect of buoyancy (force divided by mass) must be equal and opposite (ie: 9.8 meters per second per second upward). The mass of the bottle should be known. Let it be 0.1 kilograms. The buoyant force is equal to the weight of the water displaced by the bottle. Remember that weight is a force, the effect of gravity acting on a mass. The weight of the water, in newtons, is equal to its mass, in kilograms, times the effect of gravity, 9.8 meters per second per second. Each 1000 cubic centimeters (cc) of water has a mass of 1 kilogram, and thus a weight of 9.8 newtons. Knowing this, and the mass of

the bottle, we can calculate the amount of water that the bottle displaces. In other words, we can find the volume, V , of the bottle below the water line at equilibrium. Force divided by mass must equal 9.8 to balance gravity, so the equation $9.8 = 9.8 / 1000 * V / 0.1$ can be solved for V to find that 100 cc of the bottle is under water. If the bottle is 4 centimeters in diameter, we will find (from the formula for the volume of a cylinder) about 8 centimeters of its length must be below the surface.

Now suppose that the surface of the water rises suddenly. More than 8 cm of the bottle will be underwater, and the buoyant force will exceed gravity. The vertical speed of the bottle will increase and it will rise with the water. When the bottle reaches equilibrium again it will still have a positive vertical speed, so it will pass through that point and continue to rise. Now, however, it is gravity which is the larger force, and the vertical speed will be reduced until the bottle begins to descend. Eventually, these motions will disappear (due to the drag force applied by the water) and the bottle will come to rest at equilibrium. This happens so quickly that the bottle appears to be moving up and down exactly with the waves.

For larger objects, boats for example, the actual motion may be more apparent. We could treat a boat exactly like the bottle and simulate its up and down motion. As I suggested earlier, however, it is the angular rolling and pitching motion of the boat that is of real interest. To simulate these motions, we will need to know not just the total buoyant force, but also how it is distributed over the hull. The device shown in figure 2 will illustrate a general technique for finding the distribution.

You can think of this device as two bottles joined together with a stick or as the two hulls of a catamaran. Just as we calculated the buoyant force on the bottle in figure 1, we can calculate separate forces on each of these two bottles. The sum of the forces can be used to compute the vertical motion of one point on the device. This point is called the center of gravity (CG). The location of the center of gravity is critical. If you were to place the stick on a knife edge and find the point at which it balanced, this would be the center of gravity. It is the point on a body where the effect of gravity appears to be concentrated. Because the mass of an object is distributed throughout its volume, weight is a distributed force. By locating the center of gravity, however, we have a tool that transforms it into a discrete one.

We could also define a center of buoyancy, the point at which the total buoyant



Figure 1: A bottle sinks until it displaces an amount of water equal to its own weight.

with the individual parts of a body more directly to find a general method of handling force appears to act. Unfortunately, the location of this point can move significantly as the boat rises and sinks in the water. The center of gravity is also subject to some movement, such as when a passenger moves from the back to the front of the boat. Unlike the movement of the center of buoyancy, however, changes in the location of the center of gravity are not tied directly to the results of our simulation, the linear and angular motion of the boat. For this simulation, we will treat the center of gravity as stationary, and try to avoid dealing with the moving center of buoyancy.

Since we cannot deal with buoyancy as simply as we do gravity, we will have to deal

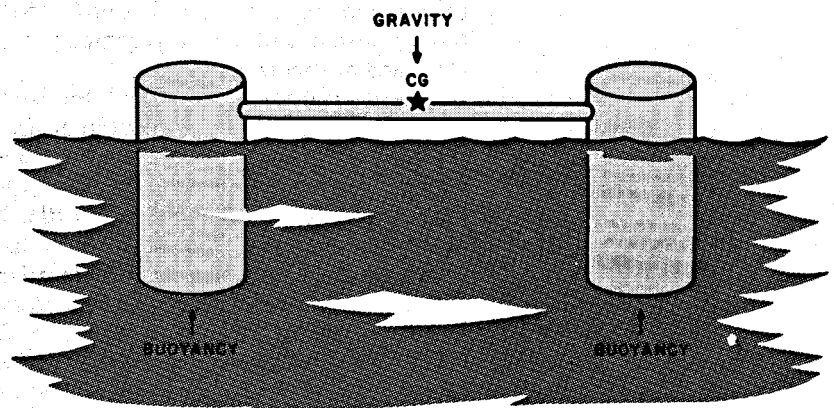


Figure 2: The distribution of the buoyant force determines the angular motion about the center of gravity.

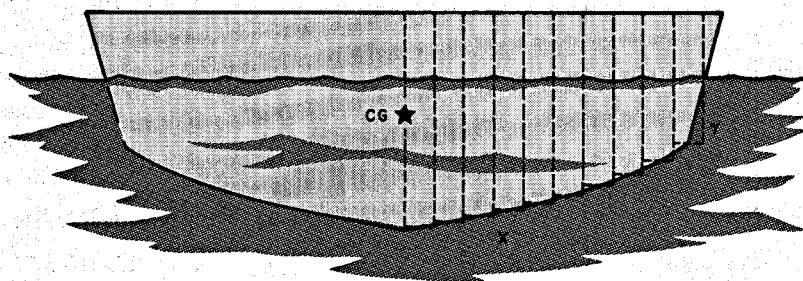


Figure 3: The continuous hull of a boat can be divided into a series of discrete segments or "bottles." X is the distance from the center of gravity to the center of the bottle. Y is the length of the bottle below the water line at equilibrium. Note that the symmetry about the CG enables us to describe the hull while only segmenting half of it.

X(m)	Y(m)
±0.05	0.50
±0.15	0.49
±0.25	0.47
±0.35	0.45
±0.45	0.42
±0.55	0.39
±0.65	0.36
±0.75	0.31
±0.85	0.37
±0.95	0.22

the distribution. In the case of the "catamaran" in figure 2, this is fairly easy. First, we assume that the bottles are small when compared to the length of the stick. Next, we assume that the center of buoyancy of each bottle is at its center, no matter how it sits in the water. Now, as far as our simulation is concerned, the entire buoyant force on the bottle acts at a point which is at a known distance from the center of gravity. This makes no difference to the vertical degree of freedom, but it is the key which allows us to simulate the angular motion.

Remember from the last article that a moment is the product of a force times a distance. In the current example, each bottle creates a moment equal to the buoyant force times the distance of the bottle from the center of gravity. Note that we define distances to the right as positive, and to the left as negative. Thus an upward force on the righthand bottle creates a positive (counterclockwise) moment. An upward force on the lefthand bottle creates a negative moment. In each simulation step, the moments are summed and then divided by the moment of inertia to find the change in angular speed each second. With this value, we can step the angular degree of freedom into the future, and return to compute new forces and moments.

Now we must determine how the combined angular and linear motion can be used to compute the new buoyant force. The force is proportional to the volume of the bottle below the waterline. For a single bottle, it was computed from the position in the vertical degree of freedom, and the location of the water surface. With the two bottle device, the vertical degree of freedom tells us only the position of the center of gravity. We must use the angular degree to find the relative position of other points on the device. If there is a positive angular position, the device will be turned counterclockwise

around the center of gravity. Consequently, the lefthand bottle will be lower than the center of gravity and the right one will be higher. The exact difference is calculated by multiplying the sine of the angular position by the distance of the bottle from the center of gravity. Again, note that points to the left have a negative distance from the center of gravity. Positive angular positions move them down.

Let's illustrate this with an example. Suppose the vertical position of the center of gravity is 0.01 meters, and the angular position is 2 degrees (0.035 radians). A bottle 1.2 meters to the left would be at

$$0.01 + \sin(0.035) * (-1.2) = -0.32 \text{ meters}$$

in other words, about 3 centimeters below its equilibrium position. A bottle 1.2 meters to the right would be

$$0.01 + \sin(0.035) * 1.2 = 0.052 \text{ meters high.}$$

The vertical position of any other point can be found similarly.

Having found the positions of the bottles, we must now find the positions of the water surface at each bottle. These will come from a sine function modified by a representative wave height, period and length. The argument of the function will be the sum of the current time divided by the period, and the bottle location (distance from the center of gravity) divided by the wave length, all multiplied by two π (to convert to radians). Once evaluated, the function is multiplied by one half the wave height (amplitude) to arrive at a final surface position. Using this scheme the surface varies with both time and location in a good approximation of sea waves.

The data in table 1 can be used to continue the example we began above. Let's place our two bottle catamaran in an inlet with 5 meter per second winds. We have determined that the water surface is given by the following formula.

$$S = \text{HEIGHT}/2 * \sin(6.28318 * (\text{TIME}/\text{PERIOD} + \text{LOCATION}/\text{LENGTH}))$$

At TIME = 1.8 seconds, we would find that the water surface at the left bottle is $0.2/2 * \sin(6.28318 * (1.8/2 + (1.2/5))) = 0.084$ meters, just below the equilibrium position. At the right bottle, the surface is at $0.2/2 * \sin(6.28318 * (1.8/2 + 1.2/5)) = 0.077$ meters. If we subtract the positions of the bottles from these values and add the 8 centimeter length of the bottles underwater at equilibrium, we will have calculated the length of each bottle below the surface at TIME = 1.8 seconds.

For the left bottle this will be $(-0.084) - (-0.032) + 0.08 = 0.28$ meters. For the right bottle this will be $0.077 - 0.052 + 0.08 = 0.105$ meters. If the bottles are 4 centimeters in diameter, then the left one displaces $0.04^2 \times 3.14159/4 \times 0.028 = 3.45 \times 10^{-5}$ cubic meters and has a buoyant force of $9800 \times 3.45 \times 10^{-5} = 0.345$ newtons. The moment it produces is $0.345 \times (-1.2) = -0.414$ newton meters. Similarly, the right bottle displaces 9.67×10^{-5} cubic meters and produces a force of 0.948 newtons and a moment of 1.14 newton meters. The sum of the forces, 1.293 newtons, is used to update the vertical degree of freedom. The sum of the moments, 0.534 newton meters, is used to update the angular degree of freedom. Now we can compute new positions, forces, moments, etc, and begin the cycle again.

Simulating the motion of a two bottle catamaran may not be very useful, but the technique is easily extended to real boats. Instead of thinking of the hull of your boat as a continuous surface, think of it as a collection of "bottles." Figure 3 shows a boat hull that has been divided in this manner. The hull can now be described by a table of Xs and Ys. The Xs represent the distances from the center of gravity to the center of each bottle. The Ys represent the lengths of each bottle below the waterline at equilibrium. Now, instead of making a series of calculations for one or two bottles, we make them for many. Just as before, the sum of the forces influences the vertical motion of the center of gravity, and the sum of the moments influences the angular motion around the center of gravity.

We now have an effective method for dealing with distributed forces. We simply divide the area over which the force acts into small segments. Within each segment, we neglect the distribution and calculate a discrete force and moment. Finally, we sum the force and moments to find the effects on the linear and angular speeds.

I have included a BASIC program with this article to illustrate the technique as applied to our boating example. With the data supplied, it computes the rolling motion of the hull cross section pictured in figure 3. Boating enthusiasts will be able to insert some other hull cross sections (deep vee, trihull, etc) in the data statements and compare the response to the sample sea states. If lateral (side to side) sections are used, the program will simulate rolling motion. If longitudinal (fore and aft) sections are used, the program will compute pitching motions. Interested readers should be able to extend the program to include three dimensional boat models and simulate

Listing 1: This program simulates the vertical and angular motion of a boat in response to sea waves. Because it involves a lengthy summation, it is inherently slow. I have, therefore, used only the second order predictor corrector formulas, and have employed a large step size. Readers who want more accuracy and who can afford to wait for results should implement the fourth order equations presented in my previous article on automotive applications (page 41). They should also increase the number of "bottles" used to describe the hull, and decrease the step size.

It should also be noted that the program does not simulate the viscous damping action of the water. As a result, if you are unfortunate enough to specify a resonant frequency of the hull as the wave period, the boat will appear to leap out of the water. While this result is obviously erroneous, it will highlight a design to be avoided.

```

100 REM SHIP MOTION SIMULATION
110 REM DESCRIBE HULL CROSS SECTION
111 REM X IS DISTANCE FROM CG TO CENTER OF BOTTLE
112 REM Y IS LENGTH OF BOTTLE BELOW WATER AT EQUILIBRIUM
120 DIM X(10),Y(10)
140 DATA 0.05,0.5,0.15,0.49,0.25,0.47,0.35,0.45,0.45,0.42
150 DATA 0.55,0.39,0.65,0.36,0.75,0.31,0.85,0.27,0.95,0.22
160 FOR J=1 TO 10
170 READ X(J),Y(J)
180 NEXT J
190 REM SET BUOYANCY FACTOR; "BOTTLE AREA"*DENSITY*9.8
200 B=0.01*1.03*9.8
205 REM COMPUTE MASS(M) & MOMENT OF INERTIA OF CROSS SECTION
210 M=0
211 I=0
212 FOR J=1 TO 10
214 M1=B/9.8*Y(J)
216 M=M+M1*2
218 I=I+M1*X(J)*2
220 NEXT J
230 REM SET SEA STATE: HEIGHT(H),LENGTH(L),PERIOD(P)
240 H=0.2
250 L=5
260 P=2
270 REM INITIALIZE INTEGRATION VARIABLES
280 DATA 0,0,0,0,0,0,0,0,0,0,0,0
290 READ Z,Z1,U,U1,A,A1,R,R1,Q,Q1,C,C1,T
300 REM INITIALIZE STEP SIZE AND PRINT INTERVAL
310 D=0.1
315 K=0
320 K1=0.1/D
321 PRINT "TIME(SEC)    VERTICAL POSITION(M)    ANGULAR POSITION(DEG)"
330 REM SUM FORCES AND MOMENTS ON THE "BOTTLES"
340 GOSUB 600
345 REM PREDICT VERTICAL MOTION
346 REM A,U,Z ARE ACCELERATION,SPEED, AND POSITION
350 A1=F/M-9.8
360 U=U1+D*A1
370 Z=Z1+D*U1
375 REM PREDICT ANGULAR MOTION
376 REM C,Q,P ARE ACCELERATION,SPEED, AND POSITION
380 C1=G/I
390 Q=Q1+D*C1
400 R=R1+D*Q1
410 REM SUM NEW FORCES AND MOMENTS FOR CORRECTOR FORMULAS
420 K=K+1
430 T=T+D
440 GOSUB 600
445 REM CORRECT VERTICAL MOTION
450 A=F/M-9.8
460 U=U1+D/2*(A+A1)
470 Z=Z1+D/2*(U+U1)
475 REM CORRECT ANGULAR MOTION
480 C=G/I
490 Q=Q1+D/2*(C+C1)
500 R=R1+D/2*(Q+Q1)
505 REM PREPARE FOR NEXT STEP
510 U1=U
520 Z1=Z
530 Q1=Q
540 R1=R
550 IF K<K1 THEN 340
560 PRINT T,Z,R*57.296
570 IF T<10 THEN 340
571 STOP
600 REM CALCULATE AND SUM FORCES AND MOMENTS ON "BOTTLES"
630 F=0
640 G=0
660 FOR J=1 TO 10
665 REM POSITIVE HALF OF HULL IS GIVEN
666 REM M IS VERTICAL POSITION OF WATER SURFACE AT BOTTLE J
667 REM W1 IS LENGTH OF BOTTLE BELOW WATER SURFACE
668 M=H/2*SIN(6.28318*(T/P+X(J)/L))
669 W1=Y(J)-Z-SIN(R)*X(J)+M
670 IF W1>0 THEN 672
671 W1=0
672 F1=B*W1
673 G1=X(J)*F1

```

Listing 1, continued:

```

675 REM MIRROR IMAGE GIVES NEGATIVE HALF
678 W=H/2*SIN(6.28318*(T/P-X(J)/L))
679 W1=Y(J)-Z+SIN(R)*X(J)+W
680 IF W1>0 THEN 682
681 W1=0
682 F2=B*W1
700 G2=-X(J)*F2
710 F=F1+F2
720 G=G1+G2
730 NEXT J
740 RETURN
750 END

```

TIME(SEC)	VERTICAL POSITION(M)	ANGULAR POSITION(DEG)
0.1	0.00147774275297	0.579343661886
0.2	0.00834856069988	2.1484430172
0.3	0.0240789139748	4.23893821533
0.4	0.0488239037315	6.19033468235
0.5	0.0789839977127	7.30318099994
0.6	0.10789765357	7.01051374777
0.7	0.127464693145	5.02510841752
0.8	0.130129740759	1.41142156817
0.9	0.110974303806	-3.40279261698
1	0.0695560412445	-8.66151908825
1.1	0.0105695497413	-13.4108553239
1.2	-0.0568528427092	-16.6916991803
1.3	-0.12088364889	-17.7238637849
1.4	-0.169478381978	-16.0583471104
1.5	-0.193013619278	-11.6821229305
1.6	-0.18645821109	-5.06270420258
1.7	-0.150554633294	2.89082493065
1.8	-0.0917208520104	10.9686349622
1.9	-0.0206887363075	17.8818895739
2	0.0503241234783	22.5634234899

both angular motions simultaneously.

With the inclusion of techniques for handling distributed forces and combined angular and linear motion, your collection of software tools for simulating motion is fairly complete. When using these tools on a personal computer, you should try above all to limit the scope of your simulations. Determine which motions really interest you and neglect or restrict the others. Divide the simulation into degrees of freedom, preferably three or less if your program is to execute with reasonable speed. Compute each force and moment individually, then apportion and sum them within the degrees of freedom. Finally, step the velocity and the position into the future. Use a small step size in your early runs, 0.01 seconds or less. Increase it to save run time only as long as your results do not change significantly. Following this procedure, and using the BASIC programming examples I have provided as a guide, you should be able to find some interesting new applications for your personal computer. ■

Simulation of Flight

F.R. Ruckdeschel

Several years ago, at a trade show in the New York City Coliseum, I saw a demonstration that simulated the flight of an airplane between two pylons. The simulation employed a cartoon-like representation of the pilot's cockpit view on a large video screen. The pilot interaction was via a joystick. When given a chance to test my own skill, I crashed.

The demonstration was impressive, particularly because it was in real time and attempted to mimic the flight of an actual flying machine. The major time cruncher in that animation was probably the display update. Today's microcomputer is capable of analogous *system simulations*.

The simulation presented here treats the flight characteristics of the model airplane using aerodynamic equations. For display purposes the model assumes zero visibility flight conditions in which the cockpit window view is replaced with an instrument panel readout and control tower communications. This type of interaction is much more technical than the simple graphics display, and perhaps more realistic.

To further enhance the realism, many standard cockpit controls have been simulated, including elevators, rudder, ailerons, flaps and throttle. These controls were individually defined and then combined into an overall flight model. The flight characteristics simulated involve several inertial and aerodynamic effects, including

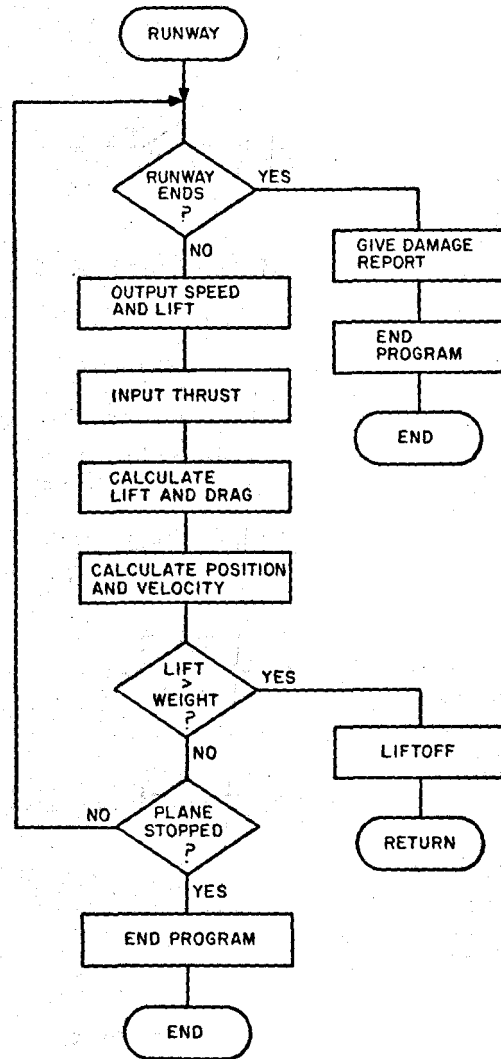
momentum, centrifugal force, air pressure, lift, drag and stall. The user chooses the basic flight characteristics which are used to represent an airplane design, ranging anywhere from a glider or Piper Cub to a jumbo jet or Phantom.

One objective of the flight simulation is to bring the plane down from a cruise altitude, 50 miles from an airport, to land on a two mile long runway (which can be shortened, if desired). The simulation portrays the landing itself, including deceleration once on the runway. Another objective is to take off from the same runway.

The extensive list of model features may best be understood by actually running the simulation or by reading through the mathematical and aerodynamical descriptions. Most users will initially have trouble flying the plane, and it may take some time to learn how to land it (probably after many crashes). The difficulty was created intentionally; flying a real airplane is not simple.

The simulation is considered in the context of a *system* model that contains various *subsystems* and *environments*, and whose output is the flight trajectory. The subsystems are the flight controls and displays. The three environments considered are takeoff, landing and flying. These three environments are linked through the executive. Transition from one to another is controlled automatically.

Figure 4: RUNWAY routine allows the pilot to maneuver the airplane on the ground. One of three conditions will cause an exit: run off end of runway, stop the airplane, lift off and start free flight.



models through simple replacement. It would be interesting to hear from readers with flight or aerodynamic experience about possible improvements and how they might affect the airplane response as perceived by the pilot.

Basic Governing Equations

Straight Line, Steady State Flight

There are several forces acting on a plane in flight. For this simulation we consider the forces of lift, drag, gravity and inertia. To make the flight behavior of the simulated airplane realistic, an approximation to the lift characteristics of an actual airfoil is used (NACA Airfoil #4412 as described in *Aerodynamics* by T von Karman, McGraw Hill (1963)):

$$L = C_L (0.4 + 5\theta_a) V^2 \quad (\theta_a < 0.28 = \theta_{amax}) \quad (1)$$

C_L is the lift coefficient, θ_a the angle of attack in radians, and v the airfoil speed. A constraint is put on the maximum angle of attack, θ_{amax} , after which the lift coefficient abruptly falls (and presumably so does the plane). For airfoil # 4412 this attack angle limit is approximately 16° (0.28 radians).

The total drag is composed of a kinetic term induced by the air disturbance related to lift, C_{DI} , and a frictional term, C_{DF} :

$$D = C_{DI}(L^2/\rho v^2) + C_{DF}(\rho v) \quad (2)$$

The normalized air density (ρ) is defined to be one at ground level.

The maximum thrust available will be described as a fraction (ξ) of the airplane weight, Mg , which is the mass of the plane (M) multiplied by the gravitational constant (g). At the maximum ground level flight speed of the airplane, v_{max} , the frictional drag is assumed to dominate, giving:

$$T_{max} = \xi Mg = C_{DF} v_{max}$$

The maximum thrust and maximum level flight ($\rho = 1$) velocity will be considered as chosen flight characteristics, so:

$$C_{DF} = (\xi Mg)/v_{max} \quad (3)$$

Another chosen characteristic is the glide angle θ_g (no power) under maximum lift conditions (flaps fully down). This corresponds to the maximum angle of attack. Under these constraints $L = (Mg) \cdot \cos(\theta_g)$ and the stall speed (v_s) and coefficient of lift, C_L , may be related using equation (1):

$$C_L = \frac{(Mg) \cos(\theta_g)}{v_s^2 (0.4 + 5\theta_{amax})} \quad (4)$$

Under these particular low altitude, no power glide conditions, assume the frictional and induced drags to be equal when the landing gear is up (i.e.: from equation (2) $C_{DI}(L^2/\rho v^2) = C_{DF}(\rho v)$). The relation between the gravitational force and drag force along the glide path is:

$$(Mg) \sin(\theta_g) = 2 C_{DF} v_s$$

or:

$$C_{DF} = \frac{(Mg) \sin(\theta_g)}{2v_s}$$

This assumption regarding the equaling of the two drags is equivalent to:

$$C_{DI} = C_{DF}(v_s)^3 / (Mg \cos(\theta_g)^2) \quad (6)$$

The variable which is key to evaluating C_L and C_{DI} is the stall speed v_s . Using equation (3) and the drag equality assumption (equation (5)) we get:

$$\xi(Mg)/v_{\max} = (Mg) \sin(\theta_g)/2v_s$$

or:

$$v_s = \frac{v_{\max} \sin(\theta_g)}{2\xi} \quad (7)$$

Equation (7) has a reasonable behavior. Assuming a plane with a thrust equal to one half of its weight (a fighter?) and a glide angle of roughly 11° (0.192 radians), then the stall speed is approximately two tenths of the maximum velocity. For a maximum velocity of 600 knots, the stall speed (flaps fully down) is calculated to be 120 knots.

To sum up, given the input parameters of plane mass, maximum thrust, and glide angle the key flight parameters required in equations (1) and (2) are obtained, as well as the stall speed.

To add to the realism of take off and final approach, the ability to manipulate flaps and landing gear is added. It is assumed that the effect of full flaps is to simply increase the airfoil lift coefficient by 50%. The effect of landing gear drag will be assumed to show up as a 60% increase in the frictional drag coefficient. Note that it is possible to create an underpowered plane that can't take off due to landing gear drag, but can fly with the gear raised. In approximation (equation (4), by inspection) we have for the relation of the stall speed with flaps to the non-flap stall speed:

$$v_s(f) = \sqrt{3/2} v_s(1)/(1+f/2)^{1/2} \quad (8)$$

where $0 \leq f \leq 1$ represents the range from no flaps to full flaps. Of course, this is really only a guess since all of the flight parameters are not known.

In the simulation, penalties are placed on stalling the plane. Also, landing without the gear down will be considered a crash situation.

Changes in Flight

Changing horizon, heading, or speed can not be done instantaneously because of inertial effects. Also, pitch oscillations are possible in an overly responsive control system. The time lags associated with such controls are a vital part of the simulation.

A change in horizon or attack angle is accomplished with the elevators. The eleva-

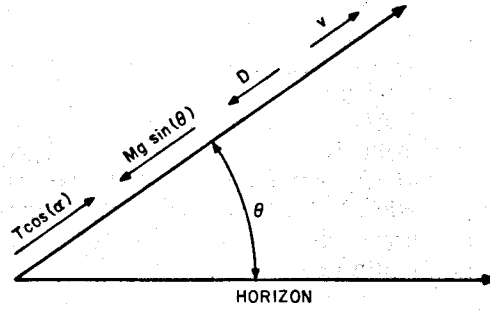


Figure 5: References used for calculating the motion along the flight path. The angle of flight is represented by θ .

tor angle (level of control) is represented by E . The rate of attack angle change ($d\alpha/dt$) is dependent on the control level and possibly plane speed. However, for this simulation assume

$$\alpha = E \quad (9)$$

The response to a change in throttle is specified in terms of an acceleration or deceleration. If the plane is in dynamic equilibrium, a change in throttle of ΔT leads to an instantaneous acceleration equal to the change in thrust divided by the mass of the airplane ($\Delta T/M$). Changes in lift cause a similar effect. This leads to the equation of motion along the flight path (see figure 5):

$$T \cos(\alpha) = Mg \sin(\theta) + D \quad (\text{constant thrust})$$

$$M \frac{dv}{dt} = T \cos(\alpha) - Mg \sin(\theta) - D \quad (\text{changing thrust})$$

or:

$$\frac{dv}{dt} = \frac{T \cos(\alpha)}{M} - g \sin(\theta) - D/M \quad (10)$$

In a time interval Δt

$$\Delta v = \frac{T \cos(\alpha)}{M} \Delta t - (g \sin(\theta) + D/M) \Delta t$$

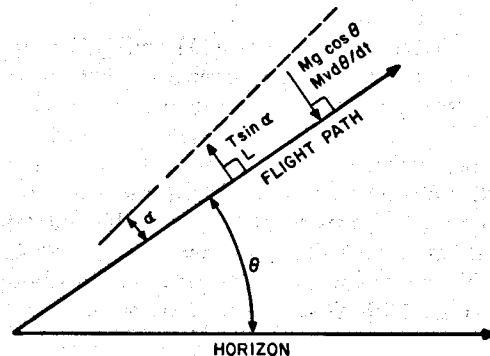


Figure 6: References used for calculating changes perpendicular to the flight trajectory. The angle of attack is represented by α .

The change in angle of climb depends on the force components perpendicular to the flight path (refer to figure 6). This includes thrust and lift.

The acceleration perpendicular to the flight path is related to the centrifugal force Mv^2/r , where r is the instantaneous loop radius. In terms of climb angle θ , the centrifugal force is $Mv \cdot d\theta/dt$. Balancing forces results in

$$L + T \sin(\alpha) - Mg \cos(\theta) = Mv \frac{d\theta}{dt}$$

or:

$$\frac{d\theta}{dt} = \frac{T}{Mv} \sin(\alpha) - \frac{g \cos(\theta)}{v} + \frac{L}{Mv} \quad (11)$$

Observe that a massive plane with a low power to mass ratio cannot change its climb angle rapidly. Also, the rate of angle change is shown to decrease with greater speed. These dependencies are intuitively reasonable.

Heading change is accomplished through a rudder and aileron control. In real life, the plane is put into a bank using the ailerons, which aid in supplying a tangential force to the trajectory. It is assumed that the pilot knows how to bank so that there is zero slip, as shown in figure 7. The addition of slip would be an interesting upgrade to the simulation, particularly in landing.

The centrifugal force is assumed to be balanced by the horizontal lift component, $L \sin(B)$, such that:

$$L \sin(B) = Mv^2/r$$

or:

$$\frac{d\rho}{dt} = \frac{L \sin(B)}{Mv} \quad (12)$$

where ρ is the heading.

It is assumed the rudder and aileron controls instantaneously result in a correct value for the bank angle B . Note how the heading change is again shown to be slower for larger and faster planes (unless the lift is increased).

Equations (9), (10), (11) and (12) completely describe the kinematic changes in attack angle, speed, climb angle, and heading. The original equation (11) was written for nonbanked flight. When banking, the L term should be replaced by the term $L \cos(B)$ since turns have a detrimental effect on lift. We can now proceed to use this information and that of the last section to establish the finite difference equations that determine the airplane's trajectory.

Finite Difference Equations

At any point in time, the position of the airplane may be described by the vector: $\vec{x} = x\hat{i} + y\hat{j} + z\hat{k}$. The z variable is the altitude of the airplane. The airport runway is specified to start at $\vec{x} = 0$ and run to $\vec{x} = +c\hat{i}$. Airport radar headings and positions are stated relative to the beginning of the runway.

The finite difference equation giving the position at time $t + \Delta t$ is:

$$\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{v}\Delta t, \quad (13)$$

Similarly:

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \frac{d\vec{v}}{dt} \Delta t. \quad (14)$$

The task is largely one of finding $d\vec{v}/dt$ and thus $\vec{v}$. The rate of velocity change (acceleration) is composed of three basic components:

- Acceleration along the flight path
- Changes in the climb angle
- Turning.

The velocity changes are composed of two parts:

- Change along the trajectory
- Change in climb angle.

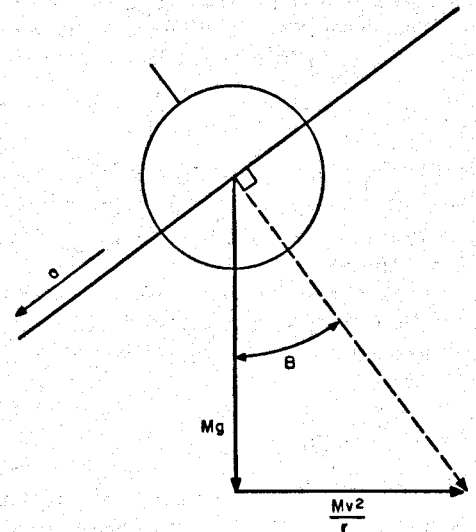


Figure 7: This simulation assumes that the airplane banks without slipping. (When banking without slip, the force vector a is zero.) The resultant force is perpendicular to the wings resulting in a change of direction.

The change along trajectory is given by:

$$\Delta \vec{v}_1 = \left(\frac{\vec{v}}{v} \right) \left\{ \frac{T \cos(\alpha)}{M} - g \sin(\theta) - \frac{D}{M} \right\} \Delta t. \quad (15)$$

The change in the climb angle is a vector which is instantaneously perpendicular to the velocity vector. For simplicity we will assume the related velocity change to have only a vertical component scaled by the cosine of the climb angle:

$$\Delta \vec{v}_2 = v \cos(\theta) \frac{d\theta}{dt} \Delta t \hat{k} = v \cos(\theta) \left\{ \frac{T}{Mv} \sin(\alpha) - \frac{g \cos(\theta)}{v} + \frac{L}{Mv} \right\} \Delta t \hat{k} \quad (16)$$

The third velocity change contribution is from the change in heading which is assumed to be only in the horizontal plane. Thus:

$$\Delta \vec{v}_3 = \left\{ \frac{\hat{k} \times \vec{v}}{\sin(\theta)} \right\} \frac{dB}{dt} \Delta t = \frac{L \sin(B)}{v \cos(\theta)} \{ v_x \hat{i} - v_y \hat{j} \} \Delta t \quad (17)$$

The notation $\hat{k} \times \vec{v}$ stands for the vector cross product. This is used as an approximation.

Combining the above equations we have:

$$\Delta v_x = \frac{\Delta t}{Mv} \left[\{ T \cos(\alpha) - Mg \sin(\theta) - D \} v_x - \frac{L \sin(B)}{\cos(\theta)} v_y \right] \quad (18a)$$

$$\Delta v_y = \frac{\Delta t}{Mv} \left[\{ T \cos(\alpha) - Mg \sin(\theta) - D \} v_y + \frac{L \sin(B)}{\cos(\theta)} v_x \right] \quad (18b)$$

$$\Delta v_z = \frac{\Delta t}{Mv} \left[\{ T \cos(\alpha) - Mg \sin(\theta) - D \} v_z + v \cos(\theta) \{ T \sin(\alpha) - Mg \cos(\theta) + L \} \right] \quad (18c)$$

Equation (18c) can be replaced by the more obvious equation:

$$v_z = v \sin(\theta).$$

Equations (18) are used in conjunction with equations (13) and (14) to give the updated velocity and position of the airplane. The constitutive equations are:

- L: = lift (equation 1)
- D: = drag (equation 2)
- α : = attack angle (equation 9)
- θ : = climb angle (equation 11)

Now that most of the theory of operation of the simulation has been covered, consider listing 1 which is the actual program being used.

The Program

Initial Conditions

The user designs an aircraft through the choice of a few simple initial flight values. The number of input parameters has been kept to a minimum by using approximate aerodynamic interrelations within the program. The parameter values suggested in the program text correspond to a small propeller driven plane carrying about four hours of fuel when at 80% throttle (thrust). Varying the following parameters will result in many interesting designs:

- **Weight of Plane:** Self-explanatory.
- **Fuel:** At the outset of the simulation

the pilot inputs the fuel load in the same units as the mass of the airplane. The fuel usage during flight versus the throttle setting is assumed to be parabolic; full throttle eats up fuel rapidly. The constants were chosen such that under *normal* conditions, full throttle can be maintained for one to two hours before the fuel runs out. However, the engine will overheat long before that occurs.

One aerodynamic effect of fuel consumption is that the airplane weight changes with time. This is reflected in the airplane going out of trim.

- **Thrust Fraction:** This is the maximum *force* which can be exerted by the engine relative to the unladen airplane weight. Generally, a higher thrust fraction results in faster response to controls and the ability to climb rapidly. The engine power is derated exponentially with altitude, as is the lift. Thus, there is a built in ceiling (maximum level flight altitude) for all designs. If a space shuttle is to be simulated, the restriction on engine power P1 must be removed in line 1480 of listing 1.

- **Maximum Speed and Glide Angle:** These are key input design parameters which strongly determine the flight characteristics of the aircraft. The stall speed increases with increasing maximum speed. Stall speed is also an increasing function of glide angle. Planes with high maximum thrust ratios (maximum thrust versus plane weight) also tend to have relatively low stall speeds. These design parameters also affect the lift and drag coefficients in nonobvious ways. Experimentation is required to create

an aircraft design which behaves well in the air. For example, choosing too low a glide angle can lead to a plane which tends to be overly responsive. An interesting classic design which might be tried is that corresponding to a Bell X1: high power; high maximum speed; poor speed and glide angle relation (must land at high speed to maintain lift). A two mile long runway might seem short in such a simulation.

- **Time Increment:** In the take off mode, the time increment value initially chosen sets the time steps for the entire take off sequence. The time increment can not be changed during a runway roll. Once in the air, the time increment can be changed as desired. Take off is not really very exciting or tricky (unless the craft is underpowered like the Spirit of St Louis), so a relatively long increment (about 5 seconds) may be used.

Internal Initializations

There are parameter initializations which occur within the program after the flight or take off option is chosen. When the flight option is chosen, the airplane's position is set to fifty miles from the airport with an associated altitude of seven miles. This can be changed by altering the values of X1, X2, and X3 on line 980 of listing 1. The initial speed is chosen to be three quarters maximum and the velocity vector is directed southeast. The velocity vector is described by the component values $S1$, $S2$, and $S3$; the speed (V) equals $\sqrt{S1^2 + S2^2 + S3^2}$. In this initialization case $S3$ (the vertical velocity component) is set equal to zero and $S1$ is equal to $S2$.

There are several other parameters which are automatically specified:

- Flaps are positioned up ($F1=0$)
- Angle of attack is zero ($T1=0$)
- Throttle is at 30% of maximum ($T=.30 \times \text{maximum thrust}$)
- Bank angle set to zero ($B=0$)
- Engine temperature set to 280°F ($T9 = 280$)
- Trim angle is 0° ($R9=0$)
- Landing gear is up.

The net result of these initial conditions for the flight option is that the pilot takes over control of an airplane which is momentarily in level flight. The craft will most likely tend to climb or descend unless the control settings are changed. Thus, it is advisable to choose a small time increment when initially taking over the controls. Once the flight is stabilized, longer time increments may be used. The controls themselves are discussed later.

The take off option also leads to an initialization routine within the program. In this case all controls except trim are generally set to their zero positions. The trim is set to -10° . The airplane is parked at the beginning of the runway with the engine temperature set to 300°F .

Take Off Controls

There are only three controls exercised during take off: thrust, flaps, and elevators.

- **Thrust:** In the user initialization of the program, a value for the maximum thrust (jet push; propeller pull) in terms of a fraction of the airplane's weight is specified. This cannot be subsequently changed during a run. The engine control which does exist is that which determines how much of this potential thrust is applied over the next time period. This fraction is represented by T . Its maximum absolute value is unity, but it may be positive or negative. If positive, the plane accelerates; if negative, it decelerates and possibly rolls backwards. A reversible pitch propeller would permit this latter type of behavior.

For an average airplane, a reasonable take off value for the thrust is generally 0.8 to 1.0. Note, however, that maintaining a thrust value of 1.0 will eventually lead to overheating of the engine. A thrust of 0.8 can be sustained indefinitely.

- **Flaps:** The effect of flaps is to increase the lift of the plane at a given speed. In doing so, the drag is also increased. The flaps may be set to between 0° and 45° , with the maximum lift occurring at 45° . If an attempt is made to lower the flaps further, they will simply peg at the full flaps (45°) limit. Under normal take off conditions, half flaps (22.5°) is usually used from the beginning of the roll. For some aircraft design choices, the use of full flaps during the take off roll may result in so much drag that the required flight speed may not be reached by the end of the runway. The quickest take off sequence is achieved by employing zero flaps at the beginning of the roll and full flaps when the required air speed is attained. However, control of the plane as it leaves the ground then becomes tricky; the plane may sharply climb, decelerate, stall (drastically lose lift), and crash.

- **Elevator Angle:** The net effect of the elevator control is to raise or lower the nose of the airplane relative to its trajectory. When rolling on the ground, the elevator control causes the nose to point up or down relative to the horizon. The airplane's response to the controls is speed-dependent. If the airplane is stationary, the elevator control has no effect.

The maximum absolute value possible for the elevator control is 45° . This corresponds to the physical angle at which the elevators may be positioned, but is *not* the resultant nose angle; the control response is always less than the control setting unless the plane is in a high speed dive.

If the elevator angle is positive, the plane will tend to nose up. During the take off period the plane should be held down by setting the elevators slightly negative until a runway speed is attained which is roughly 20 knots greater than the stall speed. The elevator control may then be pulled forward (made a few degrees positive) to lift off.

Take Off Displays

The take off controls are exercised once each time period. The corresponding data displayed are the plane's horizon, runway speed, stall speed and lift, as well as the remaining runway length and flight time. A more detailed description of these displays follows.

- **Horizon:** When the plane is on the runway, the horizon angle is identical to the angle of attack as the trajectory is horizontal. In attempting to hold the plane down using the elevator control, the horizon will be negative. Upon lift off it will very likely be positive. The desirable range of horizon angles while on the runway is between perhaps -3° (during roll) and $+5^\circ$ (at lift off).

- **Runway Speed:** Self-explanatory.

- **Stall Speed:** The displayed stall speed will not change during the take off sequence, through it may during flight. Below the stall speed it is not possible to obtain sufficient lift to leave the ground. Do not attempt lift off just above the stall speed, since that leaves little room for error. For example, a small deceleration as the plane climbs off the runway can quickly lead to a stall. A 15% or greater margin in speed is advisable.

- **Lift:** This is a cockpit display which is not commonly found in small craft. It serves as a replacement for pilot *feel*. The lift readout is also available upon landing and is useful in determining whether or not another take off may be attempted. The value shown is simply the percent of the plane's weight which is aerodynamically supported. Take off occurs when this quantity exceeds 100%.

- **Runway Left:** Self-explanatory. Running off the end of the runway is not good procedure.

- **Flight Time:** Time which has passed in the simulation time frame. This is roughly 5 to 10 per cent of real time and depends

on the time increment chosen and the operator response time.

Airborne Displays and Controls

Once the airplane is off the runway, the control messages change into two groups: cockpit display and control tower messages. These two sets of information allow the pilot to attempt level flight, turn the airplane around, and land it back on the runway.

Cockpit Display

There are fourteen flight parameters shown in the cockpit display. Some are simple reminders of control settings, while others record the aerodynamic response to these controls.

- **Altitude:** The airplane's altitude is given in feet. This is generally more than sufficient resolution for take off and flying. However, during the last stage of the approach for landing this increment is often not fine enough. In such a situation, the rate of descent and, more importantly, angle of attack and horizon angle are used.

- **Speed:** This display indicates air speed. The ground speed is not displayed and is affected by the wind. It will be observed that nosing up will usually lead to a decrease in air speed and nosing down will cause the reverse to occur. Thrust significantly affects speed.

Very high speeds can be attained in a dive. At 20% above the airplane design maximum a warning message is issued. At 40% above the maximum a wing failure abruptly occurs.

- **Stall Speed:** The stall speed is based on program computations using the design parameters chosen by the user at the beginning of the simulation. However, *accelerated stall* can occur in tight banks (turns). This is simulated in the program by making the stall speed an increasing function of the bank angle.

- **Engine Temperature:** This display gives the engine temperature in degrees Fahrenheit. Temperatures below 430°F are considered safe. Between 430°F and 450°F a warning is issued. Above 450°F the engine shuts down until it cools off. Turn on is not automatic; a thrust value (T) must be given to restart the engine. The engine temperature is calculated from the thrust used over several previous time periods. Thus it may take a while before the engine can be restarted. Needless to say, glide conditions exist in the interim.

- **Fuel:** The remaining weight of fuel is displayed in 1 pound increments. The rate of consumption depends on the square

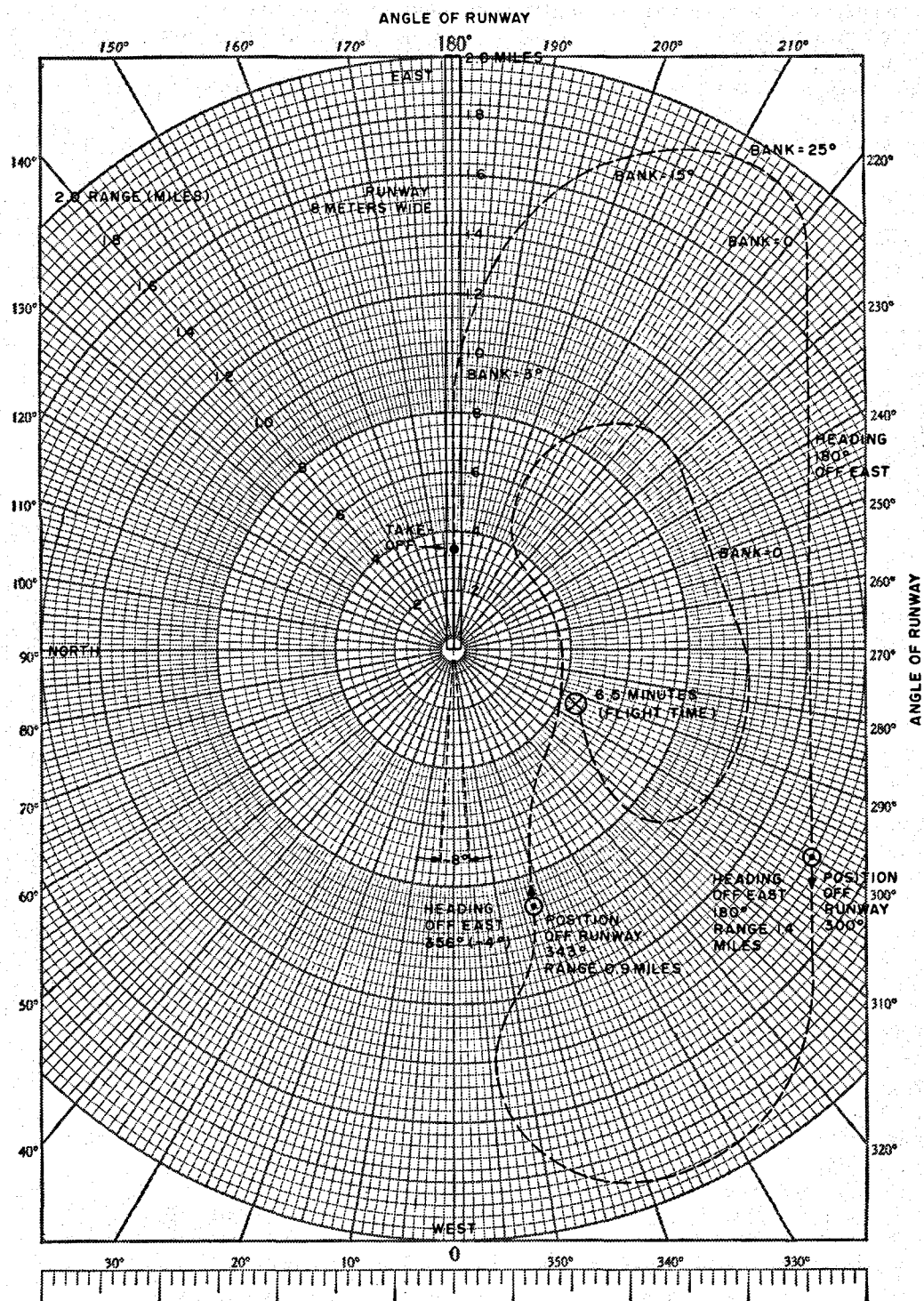


Figure 8: Polar coordinate representation of an actual computer simulation in which the course was not plotted and carefully navigated during the flight. Subsequently, the printout was used to reconstruct the flight. Toward the west end of the runway the angular position off the runway changes rapidly, leading to confusion unless the course is continuously followed on polar coordinate paper. The coordinates and headings associated with two particular positions are shown as examples. The location convention is not difficult, but may initially be somewhat confusing to those not familiar with polar coordinate geometry.

of the throttle (thrust) setting. Fuel consumption thus tends to be greatest during take off and climbs. The effect of fuel consumption on the overall plane weight is included.

- **Flaps, Trim, Thrust, Bank:** Readouts of control settings given by the pilot.

- **Attack Angle:** This generally corresponds in *sign* to the elevator angle. The magnitude of the response is speed dependent. A special condition exists when a critical angle of about 16° is exceeded. The attack angle pegs at that value. Unless in a dangerous landing approach situation, the attack angle (when flying upright) should be between roughly -3° and $+5^\circ$. When flying inverted (yes, the simulation can handle that case also), the angle of attack should be strongly negative, say -8° or more. Observe that when flying upside down, the bank, flap, trim, and elevator controls are reversed in their flight path effects (unless you are thinking invertedly also).

- **Horizon:** This display gives the angle of the horizon as it would be seen from the cockpit window. It is affected by both the climb angle and the angle of attack (a linear addition). It is an important display during final approach and landing. If the horizon is lower than -6° upon touchdown, the nose gear collapses and the plane crashes. Between -6° and 0° a nose rebound occurs, bounding the plane back into the air with the nose pointed up at an angle negatively proportional to its prior horizon. Unless care is exercised in applying the controls at this point, the plane will repeatedly bounce, nose dive, or stall: A negative horizon on touchdown causes a lot of trouble.

- **Heading Off East:** This display indicates the instantaneous direction of travel of the craft relative to due east. This particular convention was chosen as the runway runs from west (start) to east (end). When making a final approach for landing, the heading should be near 0° (between roughly 356° , -4° , and 4° in order to hit the runway properly).

- **Landing Gear, Flight Time:** These are simply status displays.

Control Tower Message

It is assumed that a radar tracking control tower exists which can aid in instrument flying by giving range, speed, and meteorological information. The following information is transmitted to the pilot:

- **Range:** This is the radial distance in miles from the west end of the runway as illustrated in figure 8. This is a key positional display.

- **Climb and Descent Rate:** This is the vertical speed of the plane in increments of one foot per second. It is an important flight indicator which is particularly useful in evaluating the net response to the elevator and throttle controls. Close to touchdown, however, the resolution of this display falls short of ideal. In this case the air speed, angle of attack, and horizon indicators become the main readouts.

- **Position Off Runway:** This readout complements the range display. It gives the angle of the plane relative to the beginning of the runway as shown in figure 8. For example, when the airplane is on or over the runway headed due East, the angle off runway is 180° . The coordinate orientation has been chosen such that the angular position upon final approach should be near 0° or 180° depending on the approach. Also, the heading off east should be near 180° or 0° , respectively (with no wind). Quantitatively, if the heading is 180° off east at a range of 400 meters (about one quarter mile), the angular position should be between 359.4° (-0.6°) and $+0.6^\circ$. This is not easy to do while also maintaining a good glide angle. Fortunately the runway is long.

- **Wind, Direction, and Speed:** The existence of a changeable wind sometimes makes landing a difficult chore. It largely affects the heading one must take in order to maintain an appropriate glide path. If the cross runway component of the wind exceeds six knots, the airplane may run off the side of the narrow runway if the heading just compensates for the wind at touchdown. Two alternatives exist: circling until the wind changes direction or diminishes, or changing heading just before touchdown to straighten out the airplane relative to the runway. Since the bank angle control works under the assumption of no slip, side slip is not simulated. If it were it could be used to handle the crosswind.

Cockpit Controls

There are quite a few controls which are exercised by the pilot during flight. Two, the time increment (S) and the continue character (C), have to do with the mechanics of the simulation. Once a control parameter is entered, it is latched until changed by the pilot. This is convenient once a quasi-stable flight pattern has been established. However, establishing a stable flight path is not easy. Constant control conditions may cause the airplane to rise, lose air speed and lift, nose over, dive, pick up air speed and lift and rise again. This cyclic pattern may be very extreme under some conditions.

The flight controls available in the cockpit are as follows:

- **Throttle (Thrust) (T):** This is the fraction of maximum thrust available, the maximum having been established in the program initialization. The entered value should be between zero and one for positive thrust, or a negative one and zero for negative thrust.

- **Bank Angle (B):** This puts the airplane into a no-slip bank (centrifugal force perpendicular to wings). A value of 180° will invert the craft; 360° will complete the roll. Absolute values greater than 360° are not allowed. Vertical lift is lost during a bank according to the cosine of the bank angle. Stall speed also increases as the turn becomes tighter. During a tight turn, the airplane will generally lose altitude even though the nose may be on the horizon. In such a situation, increased throttle and raising the nose above the horizon helps.

- **Elevators (Attack Angle) (E):** As mentioned earlier, this affects the angle of the airplane's wings relative to its trajectory (ie: the attack angle). The response is speed dependent. If an attempt is made to exceed an absolute attack angle of 16° a significant loss in lift results due to turbulence in the air flow over the wings. In effect, a stall occurs. Attack angles over 10° should be avoided.

As noted earlier, the attack angle response to the elevator control is air speed dependent. In a dive, as the air speed increases, so does the attack angle response, thus increasing lift and reducing the dive. In a climb, as the air speed decreases, the reverse happens. Thus there is some self-compensation built into the control.

- **Flaps (F):** Explained earlier. Once in flight, the flaps should be set to 0° to reduce drag.

- **Trim (R):** As far as the program is concerned, trim has an effect proportional to flaps: positive trim increases lift and negative trim decreases lift. In flight, the trim is adjusted to somewhere between -10° to $+10^\circ$ to give neutral elevator controls (ie: E set to zero gives level flight) at the chosen cruise speed and altitude. As fuel is consumed the craft will tend to rise; "trimming" will counter this. Many pilots enjoy continuously trimming their craft; it replaces nail biting.

The trim value set by the initialization is -10° . During take off this can not be altered. Once in the air the trim angle may be set to zero. However, the change should be made slowly to maintain control over the plane. Rapid changes lead to over control and erratic flight.

- **Landing Gear (G):** An input value

for G of one lowers the landing gear. Setting G to zero raises the gear. The simulated airplane has an automatic warning system which acts when the airplane is descending and is below an altitude of approximately one hundred feet. After a long flight it is not unusual to forget to lower the landing gear. This should be checked perhaps a quarter mile from touchdown so that there is time for compensation for landing gear (aerodynamic) drag.

Touchdown Conditions

Landing on the runway is a very exacting exercise since several criteria must be satisfied.

First, the landing gear must be down. Next, the airplane must be within four meters of the runway centerline. Though the runway is long, it is also narrow. The airplane must obviously also be on the runway (not short or long). The horizon should be greater than -6° ; 0° is very good.

The quality of the touchdown is finally determined by the rate of descent at contact. If less than 1.6 ft/s, the landing is considered soft. If between 1.6 and 5 ft/s, the landing is rated moderate. Between 5 and 33 ft/s touchdown is declared hard and a bounce occurs. Beyond that a crash condition exists.

One of the dangers to be wary of after a bounce is subsequently nosing over into the runway. To avoid this keep the nose up and apply a little more throttle.

Once the craft has settled on the runway and deceleration has begun, a test is made to determine if the end of the runway has been reached. If so, a crash has occurred; there are no survivors if the speed on leaving the runway was greater than 20 knots.

Deceleration is simple: reverse thrust is applied by inputting a negative throttle value. If forward thrust is used instead, the program will switch to the take off routine.

Additional Surprises

In addition to these initializations and commands there are several more variables which affect the airplane's simulated performance:

- **Wind Effects:** The effect of wind on the airplane velocity relative to ground is modelled using a random number generator. On the average, once every ten seconds the wind direction and magnitude randomly shifts (actually, there is some correlation with the previous wind vector). The net effect is that the wind vector slowly shifts with time. This effect is most important

when trying to land. In fact, under some conditions, landing may be impossible.

- **Ice Storms:** There is one chance in a hundred that in any particular time period an ice storm will develop. The consequence of such an occurrence is an increase in the airplane's weight by 50% due to ice on the wings. This obviously creates a problem if altitude can not be maintained at a safe throttle level and if the runway is too far away.

- **Altitude:** The effect of altitude on lift, drag, and engine thrust is accounted for by assuming that the air density decreases exponentially with altitude (decreases by $1/e$ approximately every 23000 feet). This automatically places a flight ceiling restriction on the particular simulation; it is not possible to escape the Earth. This feature is not an actual subroutine, but is part of other subroutines.

Command Structure

The pilot input command structure is simple. In the flight environment the computer supplies command prompts (?) and expects replies as follows:

```
? <command letter> <carriage return>
? <control value> <carriage return>
```

When on the runway, inputs for the throttle, flaps, and elevators are specifically asked for.

Program Execution

It is common to see someone spend more than an hour attempting to learn how to fly (just fly; not land) the simulated airplane. The simulation is not easy to master, like many Lunar Lander type programs. Practice is required to get the feel of the flight response of the particular airplane design chosen. In addition, the response changes with altitude due to the change in air density. I have seen some sessions last more than six hours (computer time, not flight time), eventually ending in a crash. The longest flights tend to be those in which the flying option is chosen.

This has initial conditions in which the airplane is flying at seven miles altitude 50 miles from the runway, heading in the wrong direction. The pilot must learn to navigate somewhat to make a proper landing.

Beginners usually discover quickly that it is not very difficult to accidentally stall or, if the thrust is great enough, to loop the airplane. A tendency to loop is particularly apparent for high lift or high thrust airplane designs. Recall the balsa wood (or styro-foam) gliders of your youth; when the main wing was moved forward, the airplane had a tendency to loop and stall in a cyclic fashion. A similar situation is possible under certain conditions in this simulation.

The simulation is sufficiently complete to allow not only flying loops, but also rolling and flying upside down (if the angle of attack is sufficiently negative), and perform the aerobatics normally associated with flying.

A very simple simulation run is shown in listing 2. The take off option was chosen, with the intent being to immediately land after take off using the remaining runway, and then take off again. The flight path takes one through many flight regimes, including lift off, free flight, touchdown, and runway maneuvers.

Notes

The computer simulation presented has not been completely debugged even though it has been extensively exercised. A problem with large simulations is that a bug may go undetected for some time. To aid in fixing such problems and upgrading the simulation submodels, a variables list is given in table 1.

Although the physical description of the aerodynamics of flight is not rigorous in all its subelements, the model approximations are sufficient to simulate the general interactive characteristics of flying. This philosophy of subsystem approximation for the sake of system simulation is key to the successful modeling of many systems. Quite often it is too easy to get bogged down in the details of modeling the elements only to find that the important system features may be demonstrated using less than precise inputs. This flight simulator is one such example.

Program Notes

The simulation was encoded using a subset of North Star BASIC, Version 6 Release 2. I tried to avoid using special functions which may not be available in less advanced BASIC interpreters so that the program could be easily translated into most BASICS. The statement line widths were generally kept below 40 characters because of the printout limitations of my SWTPC PR40 matrix printer.

There are a few peculiarities of North Star BASIC which must be observed in making a translation to another BASIC.

Line Delimiter: In MITS BASIC, two or more statements can be placed on one line if they are separated by a colon. North Star BASIC uses a backslash. When listing the program using Processor Technology's VDM-1 system this results in a NEW SPEED request. The VDM-1 driver can easily be changed (see manual) so that some less troublesome character prompts a display speed change.

Strings: All strings in North Star BASIC are one dimensional and, if greater than ten characters in length, must be subscripted. This is not necessary in MITS BASIC; lines 20 through 50 may be deleted and replaced with:

```
20 DIM K(12).
```

Format: In North Star BASIC, the carriage return after a print statement can be avoided by using a comma. In MITS BASIC, a semi-colon is used.

The basic operators used are fairly standard: +, -, /, *, <, =, > and ↑. The functions called are SIN, COS, SQRT, EXP, ABS and INT. The commands employed are IF-THEN, GOTO, GOSUB, RETURN, INPUT, PRINT, STOP, and REM. These capabilities are common to most BASIC interpreters. Observe that two functions are conspicuously missing: LOG and ATAN. The logarithm function is not used and the inverse tangent function is calculated in a subroutine since several BASICS, including North Star's, do not have this function.

Running the program in the form shown in listing 1 requires about 14 K bytes of program memory. Removal of all REM statements, the instruction subroutine, and the instruction string list reduces the memory requirement to about 8 K bytes. A further memory savings can be incurred if an ATAN function is used along with the commands IF-THEN-ELSE, and ON-GOTO.

Program execution is relatively fast. The majority of the time is spent printing out conditions and awaiting pilot input. The longest pause associated with actual computing (based on an IMSAI 8080 with fast memory) is less than 7 seconds. Although the program looks long and inefficient in BASIC, little would be gained by going to machine language or a compiler *unless* graphics were to be included. Graphical displays require very rapid updating routines and necessitate the use of machine language routines. A small part of the inefficiency apparent in the program is due to its user-oriented structure. All internal calculations are performed in metric units, while all IO routines use the traditional units such as knots and feet. ■

```

10 REM MICROCOMPUTER FLIGHT SIMULATOR
20 REM WRITTEN BY F.R. RUCKDESCHEL
30 REM 773 JOHN GLENN BLVD.
40 REM WEBSTER, NEW YORK 14580
50 REM VERSION 4 AS OF 1900 HOURS, 3/13/78
60 DIM K(12),R$(26),S$(14),T$(9)
70 DIM C$(11),D$(18),E$(19),F$(21),G$(14)
80 DIM H$(16),I$(14),J$(20),K$(20),L$(20)
90 DIM M$(15),N$(21),O$(32),P$(14),Q$(12)
100 P2=3.141595\PI:P3=3.28\PI:P4=57.28
110 P5=1609\PI:P6=.5148\G=9.8
120 PRINT "**** FLIGHT SIMULATOR ****"
130 PRINT "THIS PROGRAM SIMULATES FLYING,"
140 PRINT "LANDING AND TAKE-OFF"
150 GOSUB 5160
160 PRINT "DO YOU WISH TO FLY (TYPE 1)"
170 PRINT "OR TAKE-OFF (TYPE 0):",
180 T9=0
190 T4=0\PI:P1=0
200 INPUT Z
210 IF Z=1 THEN GOTO 240
220 IF Z=0 THEN GOTO 500
230 IF Z<>1 THEN GOTO 160
240 GOSUB 750
250 REM FLIGHT CHARACTERISTICS INPUT
260 GOSUB 930
270 REM INITIAL FLIGHT CONDITIONS
280 GOSUB 650
290 REM STALL SPEED CALC.
300 GOSUB 1180
310 REM CALC. OF CONSTANTS
320 IF X3<0 THEN X3=0
330 GOSUB 1810
340 REM COCKPIT DISPLAY
350 GOSUB 2950
360 REM CONTROL TOWER
370 GOTO 3520
380 REM TOUCHDOWN TEST
390 IF X3=0 THEN GOTO 4150
400 REM RUNWAY MANEUVERS
410 GOSUB 3180
420 REM PILOT INPUT
430 GOSUB 2390
440 REM ENGINE TEMP ROUTINE
450 IF T1>=16/P4 THEN GOSUB 2580
460 GOSUB 1420
470 REM NEW VELOCITY CALCULATION
480 GOTO 320
490 REM *****
500 REM TAKE-OFF EXECUTIVE
510 GOSUB 4450\GOSUB 750
520 REM FLIGHT CHARACTERISTICS INPUT
530 GOSUB 650
540 REM STALL SPEED CALC.
550 GOSUB 1180
560 REM CALC. OF CONSTANTS
570 PRINT\PRINT\PRINT
580 PRINT "READY FOR TAKE-OFF"
590 GOSUB 4500
600 REM TAKE OFF ROUTINE
610 IF L<M*G THEN GOTO 590
620 PRINT "YOU ARE IN THE AIR"
630 GOTO 320
640 REM *****
650 REM STALL SPEED CALC.
660 REM F=FLAPS
670 REM V1=MAX. SPEED
680 REM T2=GLIDE ANGLE
690 REM N1=THRUST RATIO
700 REM V2=STALL SPEED
710 V2=V1*SIN(T2)/(2*N1)
720 V2=V2*(1+.3*ABS(SIN(B)))
730 V2=V2*SQR(1.5/(1.0+F/2))\RETURN
740 REM *****
750 REM FLIGHT CHARACTERISTICS
760 PRINT"INPUT THE FOLLOWING"
770 PRINT "MASS(TONS): ",\INPUT M
780 PRINT"FUEL (TONS): ",\INPUT F9
790 PRINT"THRUST FRACTION: ",\INPUT N1
800 N1=1.25*N1
810 PRINT "MAX SPEED(KNOTS): ",
820 INPUT V1
830 PRINT"GLIDE ANGLE(DEGREES):",
840 INPUT T2\T2=ABS(T2)
850 PRINT"TIME INCR.(SEC):",\INPUT T3
860 M=M*907\PI:P1=V1*P6\T2=T2/P4
870 F9=F9*907
880 M9=M*M=M9+F9
890 PRINT\PRINT
900 PRINT"READY FOR FLIGHT"
910 PRINT\PRINT\PRINT\RETURN
920 REM *****
930 REM INITIAL FLIGHT CONDITIONS
940 REM X1,X2 AND X3 ARE POSITIONS
950 REM S1, S2 AND S3 ARE VELOCITIES
960 REM INITIAL VELOCITY IS 3/4 MAX.

```

```

970 REM INITIAL ALTITUDE IS 7 MILES
980 X1=50.032/SQR(2)\X2=-X1\X3=7
990 S1=.75*V1/SQR(2)
1000 X1=X1*P5\X2=X2*P5\X3=X3*P5
1010 S2=-S1\PI:P3=0.75*V1\PI:P4=0
1020 REM B=BANK ANGLE
1030 REM D3=CLIMB ANGLE
1040 REM G1=LANDING GEAR
1050 REM N3=THRUST
1060 F=0\PI:P1=0\N3=.3\PI:P2=280\PI:P3=0
1070 R9=-10
1080 REM T1=ANGLE OF ATTACK
1090 N2=0\PI:P3=0\G1=0\RETURN
1100 REM *****
1110 REM STALL PENALTY
1120 REM PLANE LOSES LIFT
1130 FOR J=1 TO 6
1140 L=L*V/V2
1150 NEXT J
1160 RETURN
1170 REM *****
1180 REM CALC. OF CONSTANTS
1190 REM G=GRAVITATIONAL ACCELERATION
1200 REM C1=FRICITIONAL DRAG COEF.
1210 C1=N1*M*G/V1
1220 REM C2=INDUCED DRAG COEF.
1230 C2=C1*(V2^3)/((M*G*COS(T2))^2)
1240 REM C3=LIFT COEF.
1250 C3=M*G*COS(T2)
1260 C3=C3/((V2*V2*(1.0+5*16/P4)))
1270 C3=1.5*C3
1280 RETURN
1290 REM *****
1300 REM LIFT AND DRAG CALCULATIONS
1310 REM L=LIFT D=DRAG
1320 P1=EXP(-X3/7000)
1330 IF V=0 THEN V=0.0000001
1340 L=C3*(1.+F/2)*(1.0+5*T1)*V*V
1350 L=L*P1*(1-1/(1+(V-V2)*(V-V2)))
1360 IF V<V2 THEN GOSUB 1110
1370 REM LANDING GEAR (G1) ADDED TO DRAG
1380 D=C2*L*L/(P1*V*V)+C1*V*(1+.6*G1)*P1
1390 GOSUB 650
1400 RETURN
1410 REM *****
1420 REM NEW VELOCITY CALCULATION
1430 REM FOUR POINT INTEGRATION
1440 T3=T3/4
1450 I=0
1460 I=I+1
1470 GOSUB 1300\GOSUB 1670\GOSUB 1730
1480 Z=N1*N3*M*G*COS(T1)*P1
1490 Z=Z-M*G*SIN(D3)-D
1500 Y=L*SIN(B)/COS(D3)
1510 W=T3/(M*V)
1520 S1=S1+W*(Z*S1-Y*S2)
1530 S2=S2+W*(Z*S2+Y*S1)
1540 S3=V*SIN(D3)
1550 V=SQR(S1*S1+S2*S2+S3*S3)
1560 X1=X1+S1*T3\X2=X2+S2*T3
1570 X3=X3+S3*T3
1580 IF X3<0 THEN I=4
1590 IF I<4 THEN GOTO 1460
1600 T3=T3*4
1610 REM WIND EFFECTS
1620 GOSUB 5070
1630 X1=X1+S4*T3
1640 X2=X2+S5*T3
1650 RETURN
1660 REM *****
1670 REM NOSE ANGLE CALCULATION
1680 REM T1=NOSE ANGLE REL. TO FLIGHT
1690 REM T1=ANGLE OF ATTACK ALSO
1700 IF ABS(T1)>16/P4 THEN GOSUB 2580
1710 RETURN
1720 REM *****
1730 REM CLIMB ANGLE CALCULATION
1740 T7=N1*N3*G*SIN(T1)/V
1750 T7=T7-G*COS(D3)/V+L*COS(B)/(M*V)
1760 D3=D3+T7
1770 IF D3>P2 THEN D3=D3-P2
1780 IF D3<-P2 THEN D3=D3+P2
1790 RETURN
1800 REM *****
1810 REM COCKPIT DISPLAY
1820 PRINT O$(PRINT\PRINT
1830 PRINT"ALT.: ",INT(X3*P3)," FEET"
1840 PRINT"SPEED: ",INT(V/P6)," KNOTS"
1850 GOSUB 650
1860 REM STALL SPEED CALC.
1870 IF V>=V2 THEN GOTO 1910
1880 PRINT\PRINT O$
1890 PRINT\PRINT "****STALL****"
1900 PRINT\PRINT O$
1910 PRINT"STALL SPEED: ",
1920 PRINT INT(V2/P6),

```

Listing 1: North Star BASIC listing of the flight simulator. Comments on the program are in the accompanying text box.

```

1930 PRINT " KNOTS"
1940 PRINT "ENGINE TEMP: ",INT(T9)," DEG"
1950 GOSUB 2320
1960 PRINT "FUEL",INT(2.2*F9)," LBS."
1970 PRINT "FLAPS: ",INT(100*F1)/100,
1980 PRINT " DEGREES"
1990 PRINT "TRIM: ",INT(R9*100)/100,
2000 PRINT " DEGREES"
2010 PRINT "THRUST: ",INT(100*N3)/100
2020 PRINT "BANK: ",INT(B*572.8)/10,
2030 PRINT " DEGREES"
2040 PRINT "ATTACK ANGLE: ",
2050 PRINT INT(T1*P4*10)/10,
2060 PRINT " DEGREES"
2070 PRINT "HORIZON: ",
2080 PRINT INT(10*(D3+T1)*P4)/10,
2090 PRINT " DEGREES"
2100 PRINT "HEADING OFF EAST: ",
2110 M1=S1/M2--S2
2120 GOSUB 2720
2130 GOSUB 2230
2140 PRINT "LANDING GEAR: ",
2150 IF G1=0 THEN PRINT " UP"
2160 IF G1=1 THEN PRINT " DOWN"
2170 T4=T4+T3
2180 PRINT "FLIGHT TIME: ",
2190 PRINT INT(10*T4/6)/100," MIN."
2200 GOSUB 4970
2210 GOSUB 2630\RETURN
2220 REM *****
2230 REM DIVE SPEED TEST
2240 IF V<1.2*V1 THEN RETURN
2250 PRINT\PRINT O$
2260 PRINT S$\\PRINT O$\\PRINT
2270 IF V<1.4*V1 THEN RETURN
2280 PRINT\PRINT T$\\PRINT F$
2290 PRINT C$\\PRINT D$
2300 GOTO 5830
2310 REM *****
2320 REM FUEL CONSUMPTION SUBROUTINE
2330 F9=F9-T3*(N3*2)*(N3*2)*M9/200000
2340 IF F9<0 THEN F9=0
2350 IF F9=0 THEN T=0
2360 M=M9+F9
2370 RETURN
2380 REM *****
2390 REM ENGINE TEMPERATURE
2400 FOR I=1 TO 8\\K(I)=K(I+1)\\NEXT I
2410 K(9)=N3\\T9=0\\FOR I=2 TO 9
2420 T9=T9+K(I)*(I-1)*3.2\\NEXT I
2430 T9=T9+340
2440 T9=T9*(1+P1)/2\\IF T9<75 THEN T9=75
2450 IF T9<430 THEN GOTO 2500
2460 IF T9=450 THEN GOSUB 2520
2470 IF T9<450 THEN PRINT
2480 IF T9<450 THEN PRINT "ENGINE"
2490 IF T9<450 THEN PRINT "HOT"
2500 RETURN
2510 REM *****
2520 REM ENGINE WARNING
2530 PRINT
2540 PRINT "ENGINE OVERHEAT"
2550 PRINT "POWER OFF"\\PRINT
2560 N3=0\\RETURN
2570 REM *****
2580 REM ATTACK ANGLE PASS CRITICAL
2590 FOR J=1 TO 4
2600 L=L*16/(P4*T1)
2610 NEXT I\\RETURN
2620 REM *****
2630 REM LANDING GEAR WARNING
2640 IF G1=1 THEN RETURN
2650 IF X3>30 THEN RETURN
2660 PRINT\\PRINT
2670 IF D3>0 THEN RETURN
2680 PRINT "WARNING"
2690 PRINT "LANDING WITH GEAR UP"
2700 RETURN
2710 REM *****
2720 REM INVERSE TANGENT
2730 REM APPROXIMATIONS FOR DIGITAL
2740 REM COMPUTERS
2750 REM BY CECIL HASTINGS, JR.
2760 REM PRINCETON UNIVERSITY PRESS
2770 L4=(0.0000001)^3
2780 IF ABS(M2)<L4 THEN M2=2*L4
2790 IF ABS(M1)<L4 THEN M1=2*L4
2800 X=M2/M1
2810 IF ABS(X+1)<L4 THEN X=L4-1
2820 IF X<0 THEN X=-X
2830 X=(X-1)/(X+1)\\L1=.995354
2840 L2=-.288679\\L3=.079331
2850 U2=3.14159/4+L1*X+L2*(X^3)+L3*(X^5)
2860 IF M1>0 THEN GOTO 2900
2870 IF M2>0 THEN U2=P2-U2
2880 IF M2<0 THEN U2=P2+U2
2890 GOTO 2920
2900 IF M2>0 THEN U2=U2
2910 IF M2<0 THEN U2=2*P2-U2
2920 PRINT INT(U2*572.8+.5)/10," DEG."
2930 RETURN
2940 REM *****
2950 REM CONTROL TOWER
2960 PRINT\\PRINT
2970 PRINT "CONTROL TOWER MESSAGE"
2980 PRINT O$
2990 R=SQR(X1*X1+X2*X2)
3000 PRINT "RANGE: ",INT(R/16.1)/100,
3010 PRINT " MILES"
3020 P$="DESCENT RATE: "
3030 Q$="CLIMB RATE: "
3040 IF S3<0 THEN PRINT P$,
3050 IF S3>0 THEN PRINT Q$,
3060 PRINT INT(ABS(S3)*P3),
3070 PRINT " FEET/SEC"\\M2=-X2\\M1=-X1
3080 PRINT "POSITION OFF RUNWAY: ",
3090 GOSUB 2720
3100 PRINT "WIND DIRECTION: ",
3110 M1=S4\\M2=-S5
3120 GOSUB 2720
3130 PRINT "WIND SPEED: ",
3140 PRINT INT(SQR(S4*S4+S5*S5)/P6),
3150 PRINT " KNOTS"
3160 RETURN
3170 REM *****
3180 REM COCKPIT CONTROL
3190 PRINT\\PRINT
3200 PRINT "COCKPIT CONTROL",
3210 INPUT B$
3220 REM C=CONTINUE
3230 IF B$<>"C" THEN INPUT Y1
3240 IF B$="C" THEN RETURN
3250 REM T=THROTTLE OR THRUST
3260 IF B$="T" THEN N3=Y1
3270 IF N3>1 THEN PRINT M$
3280 IF N3>1 THEN N3=1
3290 REM B=BANK ANGLE
3300 IF B$="B" THEN B=Y1
3310 IF ABS(B)>=360 THEN PRINT R$
3320 IF ABS(B)>=360 THEN GOTO 3300
3330 IF B$="B" THEN B=B/P4
3340 REM E=ELEVATORS
3350 IF B$="E" THEN T1=(Y1/P4)*(V/V1)
3360 REM S=SECONDS
3370 IF B$="S" THEN T3=Y1
3380 REM F=FLAPS
3390 IF B$="F" THEN F1=Y1
3400 IF F1<0 THEN GOTO 3210
3410 IF F1>45 THEN F1=45*ABS(F1)/F1
3420 REM R=TRIM
3430 IF B$="R" THEN R9=Y1
3440 IF ABS(R9)>10 THEN R9=10*ABS(R9)/R9
3450 F=(F1+3*R9)/75
3460 REM G=LANDING GEAR
3470 IF B$="G" THEN G1=Y1
3480 IF G1>0 THEN G1=1
3490 IF G1<0 THEN G1=0
3500 PRINT\\PRINT\\GOTO 3210
3510 REM *****
3520 REM TOUCHDOWN
3530 IF X3>0 THEN GOTO 390
3540 X3=0
3550 IF G1=1 THEN GOTO 3610
3560 PRINT\\PRINT\\PRINT
3570 PRINT C$
3580 PRINT "LANDED WITH GEAR UP"
3590 PRINT D$
3600 GOTO 5830
3610 IF ABS(X2)<4 THEN GOTO 3650
3620 PRINT\\PRINT\\PRINT C$
3630 PRINT E$\\PRINT D$\\GOTO 5830
3640 PRINT D$\\GOTO 5830
3650 IF X1<0 THEN GOTO 3620
3660 IF X1>2*5280/3.28 THEN GOTO 3620
3670 IF (D3+T1)*P4>0 THEN GOTO 3870
3680 IF (D3+T1)*P4>-6 THEN GOTO 3740
3690 PRINT\\PRINT O$
3700 PRINT C$
3710 PRINT F$
3720 PRINT D$
3730 GOTO 5830
3740 PRINT\\PRINT O$\\PRINT
3750 PRINT "NOSE WHEEL HIT FIRST"
3760 PRINT
3770 D3=D3/2
3780 S3=D3*V/2
3790 V=0.9*V
3800 X3=S3*T3
3810 PRINT "BOUNCE"
3820 PRINT\\PRINT "ALTITUDE: ",
3830 PRINT INT(X3*P3)," FEET"
3840 PRINT "CLIMB ANGLE: ",INT(10*D3*P4)/10,

```

```

3850 PRINT " DEGREES"
3860 GOTO 410
3870 IF ABS(S2)>3 THEN GOTO 3620
3880 PRINT\PRINT\PRINT
3890 PRINT "*****"
3900 PRINT "*****TOUCHDOWN*****"
3910 PRINT
3920 IF S3>-.5 THEN PRINT G$
3930 IF S3>-.5 THEN GOTO 4010
3940 IF S3>-1.5 THEN PRINT H$
3950 IF S3>-1.5 THEN GOTO 4010
3960 IF S3>-10 THEN PRINT I$
3970 IF S3>-10 THEN GOTO 4010
3980 PRINT\PRINT\PRINT
3990 PRINT C$\PRINT I$\PRINT F$
4000 PRINT D$
4010 PRINT\PRINT
4020 REM RUNWAY BOUNCE TEST
4030 IF S3>-1.5 THEN GOTO 4130
4040 PRINT "*****BOUNCE*****"
4050 S3=0.5*ABS(S3)
4060 D3=S3/V
4070 X3=S3*T3
4080 PRINT\PRINT "ALTITUDE: ",
4090 PRINT INT(X3*P3), " FEET"
4100 PRINT "CLIMB ANGLE: ",INT(10*D3*P4)/10
4110 PRINT " DEGREES"
4120 GOTO 410
4130 GOTO 390
4140 *****
4150 REM RUNWAY MANEUVERS
4160 R2=10560-P3*X1
4170 IF R2>0 THEN GOTO 4220
4180 PRINT\PRINT\PRINT C$\PRINT N$
4190 IF S1<10 THEN PRINT L$
4200 IF S1>10 THEN PRINT D$
4210 IF S1>10 THEN GOTO 5830
4220 PRINT "RUNWAY SPEED: ",
4230 PRINT INT(S1/P6), " KNOTS"
4240 PRINT INT(R2),K$
4250 PRINT"THRUST:",\INPUT N3
4260 IF ABS(N3)>1 THEN PRINT M$
4270 IF ABS(N3)>1 THEN GOTO 4250
4280 IF N3>0 THEN GOTO 590
4290 REM SWITCH TO TAKE-OFF ROUTINE
4300 N3=N3/2
4310 REM FUEL CALC.
4320 GOSUB 2320
4330 GOSUB 1300
4340 S1=S1+(N3*N1*G-ABS(D/M))*T3
4350 V=ABS(S1)
4360 X1=X1+S1*T3
4370 PRINT "LIFT (%): ",
4380 GOSUB 1300
4390 PRINT INT(100*L/(M*G))
4400 IF L>M*G THEN GOTO 620
4410 IF ABS(S1)>1/3.28 THEN GOTO 4160
4420 PRINT "LANDING COMPLETE"
4430 PRINT INT(R2),K$\GOTO 5830
4440 REM *****
4450 REM TAKE-OFF INITIALIZATION
4460 X1=0\X2=0\X3=0\S1=0\S2=0\S3=0
4470 F=0\G1=1\N3=0\D3=0\N2=0\R9=-10
4480 T9=300\T1=0\RETURN
4490 REM *****
4500 REM TAKE-OFF ROUTINE
4510 PRINT "THRUST: ",\INPUT N3
4520 IF ABS(N3)>1 THEN PRINT M$
4530 IF ABS(N3)>1 THEN GOTO 4510
4540 REM FUEL CALC.
4550 GOSUB 2320
4560 REM ENGINE TEMP CALC.
4570 GOSUB 2390
4580 PRINT "FLAPS: ",\INPUT F1
4590 IF F1<0 THEN GOTO 4580
4600 IF ABS(F1)>45 THEN F1=45*ABS(F1)/F1
4610 F=(F1+3*R9)/75
4620 PRINT "ELEVATOR DEGREES: ",\INPUT T1
4630 IF ABS(T1)>45 THEN GOTO 4620
4640 T1=T1/P4\T1=(V/V1)*T1
4650 PRINT\PRINT O$
4660 PRINT "HORIZON: ",INT(T1*P4*10)/10
4670 REM FOUR POINT INTEGRATION
4680 T3=T3/4
4690 FOR I=1 TO 4
4700 GOSUB 1300
4710 S1=S1+(N3*N1*G-D/M)*T3
4720 V=S1\X1=X1+S1*T3
4730 NEXT I\T3=T3*4
4740 PRINT "RUNWAY SPEED: ",
4750 PRINT INT(S1/P6), " KNOTS"
4760 GOSUB 650
4770 PRINT "STALL SPEED: ",
4780 PRINT INT(V2/P6), " KNOTS"
4790 REM LIFT AND DRAG CALC.
4800 GOSUB 650
4810 PRINT "LIFT (%): ",
4820 PRINT INT(100*L/(M*G))
4830 R2=10560-P3*X1
4840 IF R2>0 THEN GOTO 4870
4850 PRINT\PRINT\PRINT C$\PRINT N$
4860 PRINT D$\GOSUB 5830
4870 PRINT INT(R2),K$\T4=T4+T3
4880 PRINT "FLIGHT TIME: ",INT(T4),
4890 PRINT " SECONDS"
4900 PRINT O$\PRINT
4910 IF L<M*G THEN RETURN
4920 PRINT\PRINT\PRINT
4930 PRINT "*****LIFT OFF*****"
4940 D3=0.1*V/V1\X3=D3*V
4950 S3=T3*(L-M*G)/M
4960 RETURN
4970 REM *****
4980 REM ICE STORM
4990 IF RND(ABS(V/V1)/2)>.01 THEN RETURN
5000 M9=2.0*M9
5010 PRINT\PRINT
5020 PRINT"*****DANGER*****"
5030 PRINT"SLEET STORM"
5040 PRINT"WINGS ICING UP!!!"
5050 RETURN
5060 REM *****
5070 REM WIND EFFECTS
5080 B9=ABS((V/V1)/3)
5090 C9=ABS((V/V1)/4)
5100 B9=RND(B9)\C9=RND(C9)
5110 IF B9>.1 THEN RETURN
5120 S4=(4*(B9-.05)*V+S4)/2
5130 IF C9>.1 THEN RETURN
5140 S5=(4*(C9-.05)*V+S5)/2\RETURN
5150 REM *****
5160 REM MESSAGE LIST
5170 C$="***CRASH***"
5180 D$="***NO SURVIVORS***"
5190 E$="***MISSED RUNWAY***"
5200 F$="***LOSS OF CONTROL***"
5210 G$="TOUCHDOWN SOFT"
5220 H$="TOUCHDOWN MEDIUM"
5230 I$="TOUCHDOWN HARD"
5240 K$=" FEET OF RUNWAY LEFT"
5250 N$="RAN OFF END OF RUNWAY"
5260 L$="DAMAGE TO PLANE ONLY"
5270 M$="MAX. THRUST = 1"
5280 O$="*****"
5290 R$="MAX. ROLL IS 360 DEGREES"
5300 S$="DANGEROUS DIVE"
5310 T$="LOST WING"
5320 PRINT\PRINT O$\PRINT
5330 PRINT"***BASIC INSTRUCTIONS***"
5340 PRINT"DO YOU WISH INSTRUCTIONS?",
5350 PRINT " (Y/N): ",\INPUT B$
5360 IF B$="N" THEN RETURN
5370 PRINT\PRINT
5380 PRINT"THESE ARE TWO INITIAL FLIGHT",
5390 PRINT"T CONDITIONS. ONE STARTS T",
5400 PRINT"HE PLANE 50 MILES FROM THE",
5410 PRINT" AIRPORT AT AN ALTITUDE OF",
5420 PRINT" SEVEN MILES. THE OTHER ST",
5430 PRINT"ARTS ON THE RUNWAY. THE FL",
5440 PRINT"IGHT CHARACTERISTICS OF TH",
5450 PRINT"E PLANE ARE DEFINED BY THE",
5460 PRINT" USER. THEY ARE:"
5470 PRINT" MASS OF THE PLANE"
5480 PRINT" THRUST AS A FRACTION"
5490 PRINT" OF THE PLANE WEIGHT"
5500 PRINT" MAX. LEVEL FLIGHT SPEED"
5510 PRINT" GLIDE ANGLE AT STALL"
5520 PRINT" ELEVATOR COEF. (NOSE)"
5530 PRINT" TIME INCREMENT"
5540 PRINT\PRINT\PRINT"THESE ARE TWO ",
5550 PRINT"MESSAGE SETS. ONE IS A COC",
5560 PRINT"KPIT DISPLAY WHICH IS SELF ",
5570 PRINT"EXPLANATORY. THE OTHER IS ",
5580 PRINT"A CONTROL TOWER MESSAGE GI",
5590 PRINT"VING RANGE, DESCENT RATE A",
5600 PRINT"ND POSITION RELATIVE TO TH",
5610 PRINT"E RUNWAY. THE FLIGHT CONTR",
5620 PRINT"OL FUNCTIONS ARE:"
5630 PRINT" C=CONTINUE WITH SAME"
5640 PRINT" T=FRACTION OF MAX THRUST"
5650 PRINT" B=BANK ANGLE IN DEGREES"
5660 PRINT" E=ELEVATOR (DEGREES)"
5670 PRINT" F=FLAPS (0 TO 45 DEG.)"
5680 PRINT" R=TRIM (DEGREES)"
5690 PRINT" G=LANDING GEAR(0 UP/1 DN)"
5700 PRINT" S=NEW TIME INCREMENT"
5710 PRINT\PRINT"IT IS SUGGESTED THAT",
5720 PRINT" THE TAKE OFF OPTION BE FI",
5730 PRINT"RST CHOSEN FOR EXPERIENCE.",
5740 PRINT" A GOOD STARTING TIME INCR",
5750 PRINT"EMENT IS THREE SECS. A REA",
5760 PRINT"SONABLE STARTING PLANE WOU",

```

Listing 1, continued

```

5770 PRINT"LD BE 1 TON, FUEL .3 TONS.",
5780 PRINT" THRUST OF .3, A MAX SPEED",
5790 PRINT" OF 180 KNOTS, GLIDE ANGLE",
5800 PRINT" OF 11 DEGREES. GOOD LUCK."
5810 RETURN
5820 REM *****
5830 REM FINAL STATUS
5840 PRINT\PRINT\PRINT
5850 PRINT"FINAL FLIGHT STATUS"\PRINT
5860 PRINT"TIME OF FLIGHT: ",
5870 PRINT INT(10*T4/6)/100," MIN."

5880 PRINT"FUEL LEFT: ",INT(F9*2.2),
5890 PRINT " LBS."
5900 PRINT"FINAL SPEED: ",INT(V/P6),
5910 PRINT " KNOTS"
5920 PRINT"FINAL HORIZON: ",
5930 PRINT INT((D3+T1)*P4)," DEGREES"
5940 PRINT"TRY AGAIN? (Y/N): ",
5950 INPUT B$IF B$<>"N" THEN GOTO 160
5960 PRINT"GOODBYE. COME AGAIN SOON."
5970 END
READY

```

Table 1: This table of variables and parameter definitions will aid in updating the simulation in listing 1.

B	=	bank angle (radians)
B9	=	random number in wind effects routine
C1	=	frictional drag coefficient
C2	=	induced drag coefficient
C3	=	lift coefficient
C9	=	random number in wind effects routine
D	=	drag (Newtons)
D3	=	climb angle (radians)
F	=	total flaps, including trim (normalized; 45° = 1)
F1	=	flaps
F9	=	fuel supply (kilograms)
G	=	gravitational constant (meters/second ²)
G1	=	landing gear index (0 or 1)
K(I)	=	thrust sequence
L	=	lift (newtons)
L1	=	constant in ATAN routine
L2	=	constant in ATAN routine
L3	=	constant in ATAN routine
L4	=	small constant very near zero
M	=	total airplane mass (kilograms)
M1	=	dummy variable passed to ATAN
M2	=	dummy variable passed to ATAN
M9	=	initial plane (minus fuel) mass (kilograms)
N1	=	maximum thrust ratio (normalized)
N3	=	fraction of available power used (normalized)
P1	=	air density (normalized)
P2	=	π
P3	=	feet to meter conversion factor
P4	=	degrees to radian conversion factor
P5	=	meters to mile conversion factor
P6	=	(meters/second) to knots conversion factor
Q1	=	elevator control (radians)
R	=	range of plane from runway (meters)
R2	=	length of runway left (feet)
R9	=	trim (radians)
S1	=	eastward speed component (meters/second)
S2	=	northward speed component (meters/second)
S3	=	vertical speed component (meters/second)
S4	=	eastward wind component (meters/second)
S5	=	northward wind component (meters/second)
T1	=	attack angle (radians)
T2	=	glide angle (radians)
T3	=	time increment (seconds)
T4	=	accumulated time (seconds)
T7	=	dummy variable
T9	=	engine temperature (°F)
U2	=	angle (radians)
V	=	line of flight speed (meters/second)
V1	=	maximum sea level speed (meters/second)
V2	=	stall speed
W	=	dummy variable
X	=	dummy variable in ATAN routine
X1	=	east coordinate (meters)
X2	=	north coordinate (meters)
X3	=	altitude (meters)
Y	=	dummy variable
Y1	=	dummy input variable
Z	=	dummy variable


```

RUN

*** FLIGHT SIMULATOR ***
THIS PROGRAM SIMULATES FLYING
LANDING AND TAKE-OFF

*****

***BASIC INSTRUCTIONS***
DO YOU WISH INSTRUCTIONS? (Y/N): ?N
DO YOU WISH TO FLY (TYPE 1)
OR TAKE-OFF (TYPE 0):?0
INPUT THE FOLLOWING
MASS(TONS): ?1
FUEL (TONS): ? 3
THRUST FRACTION: ? 3
MAX SPEED(KNOTS): ?180
GLIDE ANGLE(DEGREES):?11
TIME INCR.(SEC):?3

```

READY FOR FLIGHT

```

READY FOR TAKE-OFF
THRUST: ?1
FLAPS: ?0
ELEVATOR DEGREES: ?0

```

```

*****
HORIZON: 0
RUNWAY SPEED: 20 KNOTS
STALL SPEED: 61 KNOTS
LIFT (%): 0
10494 FEET OF RUNWAY LEFT
FLIGHT TIME: 3 SECONDS
*****

```

```

THRUST: ?1
FLAPS: ?25
ELEVATOR DEGREES: ?0

```

```

*****
HORIZON: 0
RUNWAY SPEED: 37 KNOTS
STALL SPEED: 53 KNOTS
LIFT (%): 1
10336 FEET OF RUNWAY LEFT
FLIGHT TIME: 6 SECONDS
*****

```

```

THRUST: ?1
FLAPS: ?25
ELEVATOR DEGREES: ?0

```

```

*****
HORIZON: 0
RUNWAY SPEED: 51 KNOTS
STALL SPEED: 53 KNOTS
LIFT (%): 29
10099 FEET OF RUNWAY LEFT
FLIGHT TIME: 9 SECONDS
*****

```

```

THRUST: ?1
FLAPS: ?25
ELEVATOR DEGREES: ?0

```

```

*****
HORIZON: 0
RUNWAY SPEED: 61 KNOTS
STALL SPEED: 53 KNOTS
LIFT (%): 76
9804 FEET OF RUNWAY LEFT
FLIGHT TIME: 12 SECONDS
*****

```

```

THRUST: ?1

*ENGINE*
***HOT***
FLAPS: ?25
ELEVATOR DEGREES: ?8

```

```

*****
HORIZON: 2.7
RUNWAY SPEED: 66 KNOTS
STALL SPEED: 53 KNOTS
LIFT (%): 116
9475 FEET OF RUNWAY LEFT
FLIGHT TIME: 15 SECONDS
*****

```

```

***LIFT OFF***
YOU ARE IN THE AIR
*****

```

```

ALT : 4 FEET
SPEED: 66 KNOTS
STALL SPEED: 53 KNOTS
ENGINE TEMP: 436 DEG
FUEL 591 LBS.
FLAPS: 25 DEGREES
TRIM: -5 DEGREES
THRUST: 1
BANK: 0 DEGREES
ATTACK ANGLE: 2.7 DEGREES
HORIZON: 4.8 DEGREES
HEADING OFF EAST: 0 DEG
LANDING GEAR: DOWN
FLIGHT TIME: .3 MIN.

```

```

CONTROL TOWER MESSAGE
*****
RANGE: .2 MILES
CLIMB RATE: 15 FEET/SEC
POSITION OFF RUNWAY: 180 DEG
WIND DIRECTION: 45 DEG
WIND SPEED: 0 KNOTS

```

```

COCKPIT CONTROL?T
?7

```

```

?E
?0

```

?C

```

*ENGINE*
***HOT***
*****

```

```

ALT : 9 FEET
SPEED: 66 KNOTS
STALL SPEED: 53 KNOTS
ENGINE TEMP: 437 DEG
FUEL 590 LBS.
FLAPS: 25 DEGREES
TRIM: -5 DEGREES
THRUST: .7
BANK: 0 DEGREES
ATTACK ANGLE: 0 DEGREES
HORIZON: -.1 DEGREES
HEADING OFF EAST: 0 DEG.
LANDING GEAR: DOWN
FLIGHT TIME: .35 MIN

```

```

CONTROL TOWER MESSAGE
*****
RANGE: .26 MILES
DESCENT RATE: 0 FEET/SEC
POSITION OFF RUNWAY: 180 DEG
WIND DIRECTION: 0 DEG
WIND SPEED: 0 KNOTS

```

```

COCKPIT CONTROL?E
?7

```

```

?T
?6

```

?C

```

*ENGINE*
***HOT***
*****

```

```

ALT : 7 FEET
SPEED: 64 KNOTS
STALL SPEED: 53 KNOTS
ENGINE TEMP: 434 DEG
FUEL 590 LBS.
FLAPS: 25 DEGREES
TRIM: -5 DEGREES
THRUST: .6
BANK: 0 DEGREES
ATTACK ANGLE: .6 DEGREES
HORIZON: - 2 DEGREES

```

Listing 2: Part of a typical session at a terminal with the simulator. Notice that the engines will begin to overheat if full throttle is applied for extended periods of time. It took several attempts to land the airplane without bouncing it. The nose must be pointed up or exactly parallel with the runway for the airplane to land safely. This output shows that even someone who has been using the simulator for some time (an experienced pilot) can have some difficulty controlling the airplane's motions.

Listing 2, continued

HEADING OFF EAST: 0 DEG.
LANDING GEAR: DOWN
FLIGHT TIME: 4 MIN.

CONTROL TOWER MESSAGE

RANGE: .33 MILES
DESCENT RATE: 1 FEET/SEC
POSITION OFF RUNWAY: 180 DEG
WIND DIRECTION: 0 DEG.
WIND SPEED: 0 KNOTS

COCKPIT CONTROL?E
7.8

?C

ENGINE
HOT

ALT.: 0 FEET
SPEED: 64 KNOTS
STALL SPEED: 53 KNOTS
ENGINE TEMP: 430 DEG
FUEL 589 LBS
FLAPS: 25 DEGREES
TRIM: -5 DEGREES
THRUST: .6
BANK: 0 DEGREES
ATTACK ANGLE: .7 DEGREES
HORIZON: -1.7 DEGREES
HEADING OFF EAST: 0 DEG
LANDING GEAR: DOWN
FLIGHT TIME: .45 MIN.

CONTROL TOWER MESSAGE

RANGE: .37 MILES
DESCENT RATE: 4 FEET/SEC
POSITION OFF RUNWAY: 180 DEG
WIND DIRECTION: 0 DEG
WIND SPEED: 0 KNOTS

NOSE WHEEL HIT FIRST

BOUNCE

ALTITUDE: 3 FEET
CLIMB ANGLE: 1.2 DEGREES

COCKPIT CONTROL?E
72

?C

ALT.: 3 FEET
SPEED: 61 KNOTS
STALL SPEED: 53 KNOTS
ENGINE TEMP: 423 DEG
FUEL 589 LBS
FLAPS: 25 DEGREES
TRIM: -5 DEGREES
THRUST: .6
BANK: 0 DEGREES
ATTACK ANGLE: 2 DEGREES
HORIZON: 2.5 DEGREES
HEADING OFF EAST: 0 DEG.
LANDING GEAR: DOWN
FLIGHT TIME: .5 MIN.

CONTROL TOWER MESSAGE

RANGE: .44 MILES
CLIMB RATE: 1 FEET/SEC
POSITION OFF RUNWAY: 180 DEG
WIND DIRECTION: 0 DEG.
WIND SPEED: 0 KNOTS

COCKPIT CONTROL?T
7.5

?C

ALT : 0 FEET
SPEED: 59 KNOTS
STALL SPEED: 53 KNOTS
ENGINE TEMP: 415 DEG
FUEL 589 LBS
FLAPS: 25 DEGREES
TRIM: -5 DEGREES
THRUST: .5
BANK: 0 DEGREES
ATTACK ANGLE: 2 DEGREES
HORIZON: -.8 DEGREES
HEADING OFF EAST: 0 DEG.
LANDING GEAR: DOWN
FLIGHT TIME: .55 MIN.

CONTROL TOWER MESSAGE

RANGE: .49 MILES
DESCENT RATE: 4 FEET/SEC
POSITION OFF RUNWAY: 180 DEG
WIND DIRECTION: 0 DEG.
WIND SPEED: 0 KNOTS

NOSE WHEEL HIT FIRST

BOUNCE

ALTITUDE: 3 FEET
CLIMB ANGLE: 1.3 DEGREES

COCKPIT CONTROL?E
72.5

?C

ALT : 0 FEET
SPEED: 59 KNOTS
STALL SPEED: 53 KNOTS
ENGINE TEMP: 409 DEG
FUEL 588 LBS.
FLAPS: 25 DEGREES
TRIM: -5 DEGREES
THRUST: .5
BANK: 0 DEGREES
ATTACK ANGLE: 2.5 DEGREES
HORIZON: -.7 DEGREES
HEADING OFF EAST: 0 DEG.
LANDING GEAR: DOWN
FLIGHT TIME: .6 MIN.

CONTROL TOWER MESSAGE

RANGE: .52 MILES
DESCENT RATE: 5 FEET/SEC
POSITION OFF RUNWAY: 180 DEG.
WIND DIRECTION: 0 DEG.
WIND SPEED: 0 KNOTS

NOSE WHEEL HIT FIRST

BOUNCE

ALTITUDE: 4 FEET
CLIMB ANGLE: 1.5 DEGREES

COCKPIT CONTROL?E
73

?C

ALT : 0 FEET
SPEED: 58 KNOTS
STALL SPEED: 53 KNOTS
ENGINE TEMP: 404 DEG
FUEL 588 LBS
FLAPS: 25 DEGREES
TRIM: -5 DEGREES
THRUST: .5
BANK: 0 DEGREES

ATTACK ANGLE: 3 DEGREES
HORIZON: .7 DEGREES
HEADING OFF EAST: 0 DEG
FLIGHT TIME: .65 MIN

CONTROL TOWER MESSAGE

RANGE: .55 MILES
DESCENT RATE: 3 FEET/SEC
POSITION OFF RUNWAY: 180 DEG
WIND DIRECTION: 0 DEG.
WIND SPEED: 0 KNOTS

TOUCHDOWN

TOUCHDOWN MEDIUM

RUNWAY SPEED: 58 KNOTS
7624 FEET OF RUNWAY LEFT
THRUST: 70
LIFT (%): 16
RUNWAY SPEED: 43 KNOTS
7402 FEET OF RUNWAY LEFT
THRUST: 70
LIFT (%): 3
RUNWAY SPEED: 36 KNOTS
7216 FEET OF RUNWAY LEFT
THRUST: 7-1
LIFT (%): 0
RUNWAY SPEED: 20 KNOTS
7114 FEET OF RUNWAY LEFT
THRUST: 7-1
LIFT (%): 0
RUNWAY SPEED: 6 KNOTS
7082 FEET OF RUNWAY LEFT
THRUST: 71
FLAPS: 225
ELEVATOR DEGREES: 70

HORIZON: 0
RUNWAY SPEED: 25 KNOTS
STALL SPEED: 53 KNOTS
LIFT (%): 0
6987 FEET OF RUNWAY LEFT
FLIGHT TIME: 42 SECONDS

THRUST: 71
FLAPS: 225
ELEVATOR DEGREES: 70

HORIZON: 0
RUNWAY SPEED: 42 KNOTS
STALL SPEED: 53 KNOTS
LIFT (%): 4
6804 FEET OF RUNWAY LEFT
FLIGHT TIME: 45 SECONDS

THRUST: 71

ENGINE
HOT
FLAPS: 225
ELEVATOR DEGREES: 70

HORIZON: 0
RUNWAY SPEED: 55 KNOTS
STALL SPEED: 53 KNOTS
LIFT (%): 54
6548 FEET OF RUNWAY LEFT
FLIGHT TIME: 48 SECONDS

THRUST: 71

ENGINE
HOT
FLAPS: 225
ELEVATOR DEGREES: 70

HORIZON: 0
RUNWAY SPEED: 63 KNOTS
STALL SPEED: 53 KNOTS
LIFT (%): 83
6240 FEET OF RUNWAY LEFT
FLIGHT TIME: 51 SECONDS

THRUST: 71

ENGINE
HOT
FLAPS: 225
ELEVATOR DEGREES: 78

HORIZON: 2.8
RUNWAY SPEED: 68 KNOTS
STALL SPEED: 53 KNOTS
LIFT (%): 123
5902 FEET OF RUNWAY LEFT
FLIGHT TIME: 54 SECONDS

LIFT OFF
YOU ARE IN THE AIR

ALT.: 4 FEET
SPEED: 68 KNOTS
STALL SPEED: 53 KNOTS
ENGINE TEMP: 445 DEG
FUEL 580 LBS.
FLAPS: 25 DEGREES
TRIM: -5 DEGREES
THRUST: 1
BANK: 0 DEGREES
ATTACK ANGLE: 2.8 DEGREES
HORIZON: 5 DEGREES
HEADING OFF EAST: 0 DEG
LANDING GEAR: DOWN
FLIGHT TIME: .95 MIN.

CONTROL TOWER MESSAGE

RANGE: .88 MILES
CLIMB RATE: 22 FEET/SEC
POSITION OFF RUNWAY: 180 DEG
WIND DIRECTION: 0 DEG.
WIND SPEED: 0 KNOTS

EXPERIMENTATION

Robot Simulation on Microcomputers

Why build a mechanism when a video display
can be used to visualize logical problems of robots?

John Webster

A short time ago I came across a program (*Dr Dobb's Journal*, volume 1, number 8, page 28) written by Marvin Winzenread, entitled "The Bouncing Beastie: A Random Walker for Processor Tech's VDM-1." When I loaded the program, sure enough, this little thimble with feet appeared and began stumbling around the screen leaving a trail of asterisks. That was it. No big deal, you say? Marvin says the little character looks like a turtle to him, but to me it looked like the robot I'd tried to build when I was in high school, and I was entranced.

I'm one of those people who has always wanted to build a robot, but never quite did it for various reasons.

Part of the problem is that once you figure out approximately *how* to build, say, a simple little robot that will wander around your living room bouncing off furniture, then there isn't really much point in going to all the trouble and expense of actually building it. You already know what it's going to do, and it wouldn't really be all *that* useful.

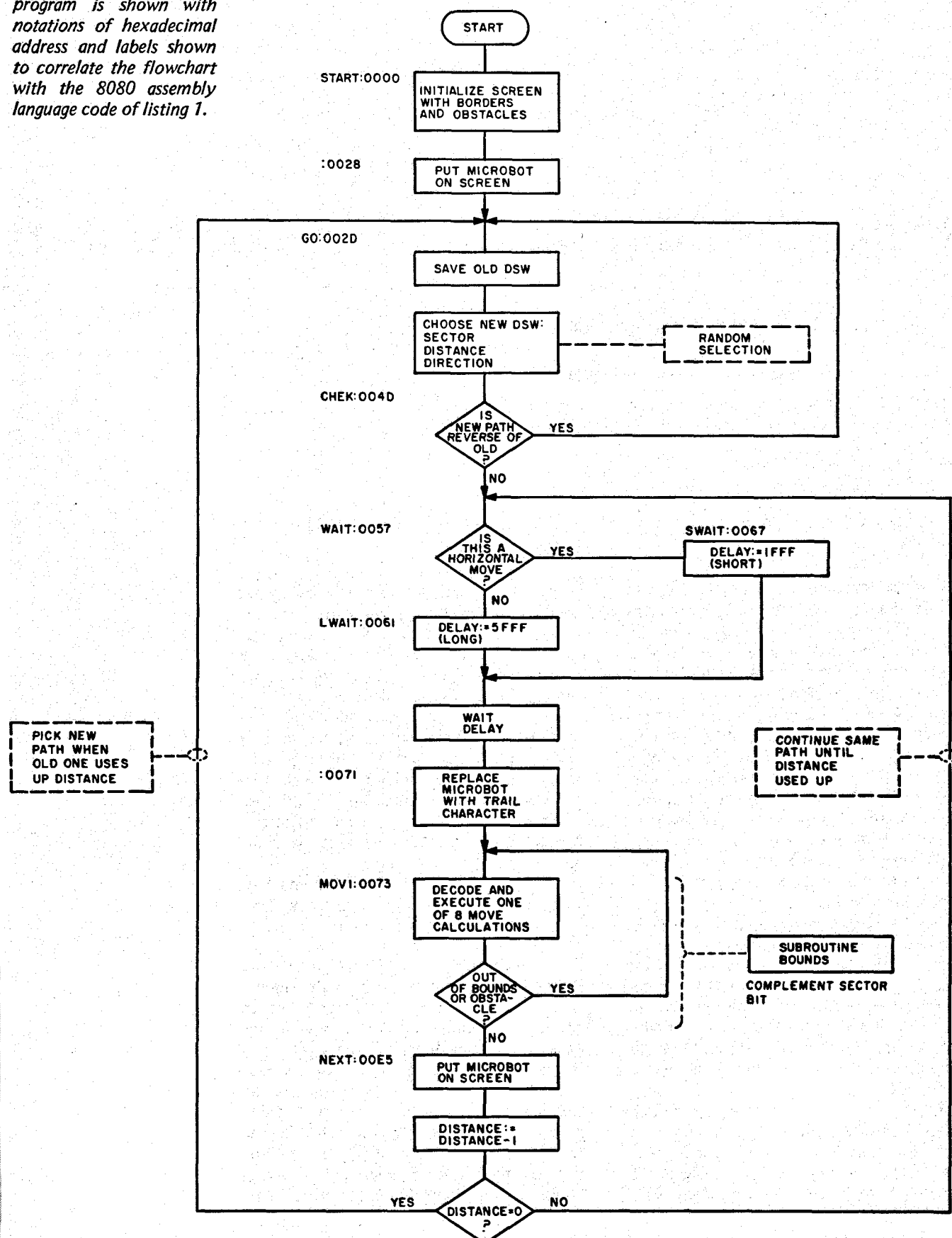
Another factor is that the hardware robots will need already exists and isn't likely to change all that much in the foreseeable future. The thing that's lacking is the software. If we had the proper programs we would all have robots today.

Watching Marvin's surprisingly fascinating little creature bounce around the screen, it occurred to me that a very interesting, entertaining and hopefully useful alternative to actually building robots might be to simulate them on a microcomputer television display. Thus, my microbots were born. Microbots are simulated robots that move about a simulated room (the television screen) under software control. With microbots you can spend your time thinking up new things for them to do and testing robot related software without worrying about the hardware.

"Microbot" is a simple 8080 machine language microbot driver program that provides a starting point for robot related program development.

In its present form (see figure 1 and listing 1) the program begins with a microbot creature starting in the center of an empty room. It then makes a move in one of eight random directions for a randomly chosen distance (of one to four squares). The microbot will sense the perimeter wall and reject any random moves that would make it walk "into" the wall. It will also reject a move that is in the opposite direction of the previous one. This prevents "bouncing" and produces a more realistic movement pattern.

Figure 1: Flowchart of the Microbot program. The program is shown with notations of hexadecimal address and labels shown to correlate the flowchart with the 8080 assembly language code of listing 1.



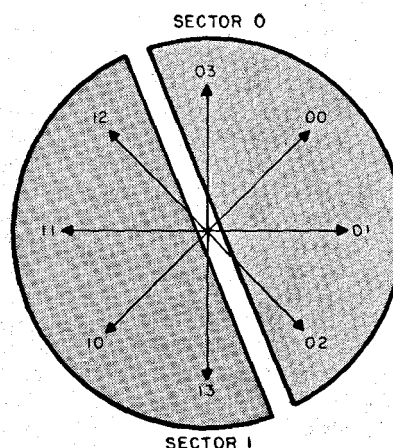
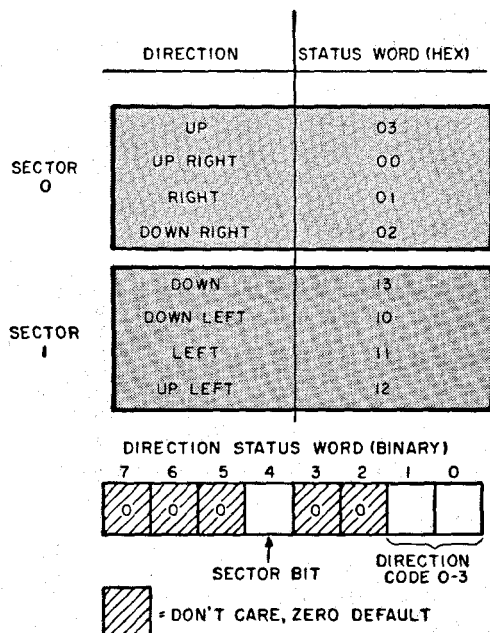


Figure 2: In the representation used for direction status words (DSWs) in the Microbot program, the possible directions break down into two sectors based on the status of bit 4 of the DSW byte. The two low order bits determine the direction within each sector according to the diagram.

A call to a random number generator (RND) first selects the 180° sector for the next move by setting bit 4 of the present direction status word (DSW) to either one or zero. The subroutine REC stores this information at hexadecimal location 14D and moves the previous contents of 14D (the DSW for the previous move) to 14E for later comparison.

A second call to the random number generator provides one of the four possible moves within the chosen sector. This value (00, 01, 02, or 03 hexadecimal) is then added to the data in 14D to produce the final direction status word for the present move.

Figure 2 illustrates the eight possible moves and their associated status words. Notice that opposite directions are indicated by the state of bit 4 of the direction status word. This fact is used in the present program to disallow immediate reverse moves. It also makes it simple to reverse direction when desired by the programmer.

There are three status words used in the present program. Location 14D contains the present direction status word; location 14E contains the direction status word of the previous move; location 14F contains a distance status word (DIS) that determines the length of the present move.

The program segment CHEK then compares the present and previous direction status words to see if they are in opposite directions. If they are, a new direction status word is sought.

Next, wait routines are inserted to slow the movement to a realistic speed. The speed of your microbot can be varied by changing the constants at hexadecimal locations 63

and 69. WAIT checks to see if the present move is horizontal or not. A longer wait is used for horizontal moves to compensate for the fact that there are 64 horizontal and only 16 vertical positions on the VDM-1 display.

A "trail" character is then inserted on the present square and the direction status word is decoded with a resulting jump to one of eight move routines. Any noncursor character may be inserted at location 72 as a trail character. After completing the proper move, the subroutine BOUNDS is called to determine whether or not the microbot is trying to move to an obstacle position (ie: any square containing a cursor). If it is, bit 4 is complemented and the resulting move returns microbot to his previous position, effectively cancelling the disallowed move. Each move routine ends with a jump to NEXT which writes the microbot character into the new position square and checks the distance status word. Depending on the contents of DIS, either the last move is repeated or the program jumps back to GO to initiate a new random move.

Future Expansion

As the purpose of this article is to present a basic concept and program for others to expand, let's look at some possible areas of development:

1. **Search Efficiency:** As many robot systems will incorporate search or random search routines, the VDM display allows an excellent theatre for study of the efficiency of various

0000	31	57	01	START	LXI	SP.0157H	00A1	19			DAD	D	
0003	AF				XRA	A	00A2	CD	F6	00	CALL	BOUNDS	
0004	D3	C8			OUT	0C8H	00A5	C3	E5	00	JMP	NEXT	
0006	16	0E			MVI	D.0EH	00A8	23			INX	H	
0008	21	00	CC		LXI	H.0CC00H	00A9	CD	F6	00	CALL	BOUNDS	
000B	01	42	A0		LXI	B.0A042H	00AC	C3	E5	00	JMP	NEXT	
000E	CD	08	01		CALL	LOAD	00AF	11	42	00	DR	LXI	D.042H
0011	01	3C	20	SCRN	LXI	B.0203CH	00B2	19			DAD	D	
0014	CD	08	01		CALL	LOAD	00B3	CD	F6	00	CALL	BOUNDS	
0017	01	40	A0		LXI	B.0A040H	00B6	C3	E5	00	JMP	NEXT	
001A	CD	08	01		CALL	LOAD	00B9	11	40	00	DN	LXI	D.040H
001D	15				DCRD		00BC	19			DAD	D	
001E	BA				CMPD		00BD	CD	F6	00	CALL	BOUNDS	
001F	C2	11	00		JNZ	SCRN	00C0	C3	E5	00	JMP	NEXT	
0022	01	3E	A0		LXI	B.0A03EH	00C3	11	3E	00	DL	LXI	D.03EH
0025	CD	08	01		CALL	LOAD	00C6	19			DAD	D	
0028	21	20	CE		LXI	H.0CE20H	00C7	CD	F6	00	CALL	BOUNDS	
002B	36	07			MVI	M.07H	00CA	C3	E5	00	JMP	NEXT	
002D	CD	1A	01	GO	CALL	RND	00CD	2B			LFT	DCX	H
0030	E6	02			ANI	02H	00CE	CD	F6	00	CALL	BOUNDS	
0032	CA	37	00		JZ	RC	00D1	C3	E5	00	JMP	NEXT	
0035	3E	10			MVI	A.010H	00D4	11	BE	FF	UL	LXI	D.0FFBEH
0037	CD	10	01	RC	CALL	REC	00D7	19			DAD	D	
003A	CD	1A	01	DIS	CALL	RND	00D8	CD	F6	00	CALL	BOUNDS	
003D	32	4F	01		STA	014FH	00DB	C3	E5	00	JMP	NEXT	
0040	CD	1A	01	DIR	CALL	RND	00DE	11	C0	FF	UP	LXI	D.0FFCOH
0043	47				MOV	B.A	00E1	19			DAD	D	
0044	3A	4D	01		LDA	014DH	00E2	CD	F6	00	CALL	BOUNDS	
0047	E6	10			ANI	010H	00E5	36	07		NEXT	MVI	M.07H
0049	80				ADD	B	00E7	3A	4F	01	LDA	014FH	
004A	32	4D	01		STA	014DH	00EA	3D			DCR	A	
004D	47			CHEK	MOV	B.A	00EB	32	4F	01	STA	014FH	
004E	3A	4E	01		LDA	014EH	00EE	FE	FF		CPI	OFFH	
0051	A8				XRA	B	00F0	CA	2D	00	JZ	GO	
0052	FE	10			CPI	010H	00F3	C3	57	00	JMP	WAIT	
0054	CA	40	00		JZ	DIR	00F6	7E			BOUNDS	MOV	A.M
0057	3A	4D	01	WAIT	LDA	014DH	00F7	E6	80		ANI	080H	
005A	E6	0F			ANI	0FH	00F9	C8			RZ		
005C	FE	01			CPI	01H	00FA	33			INX	6	
005E	CA	67	00		JZ	SWAIT	00FB	33			INX	6	
0061	01	FF	5F	LWAIT	LXI	B.5FFFFH	00FC	3A	4D	01	LDA	014DH	
0064	C3	6A	00		JMP	DEC	00FF	0E	10		MVI	C.010H	
0067	01	FF	1F	SWAIT	LXI	B.1FFFFH	0101	A9			XRA	C	
006A	0B			DEC	DCXB		0102	32	4D	01	STA	014DH	
006B	78				MOV	A.B	0105	C3	73	00	JMP	MOV	I
006C	FE	00			CPI	00	0108	70			LOAD	MOV	M.B
006E	C2	6A	00		JNZ	DEC	0109	23			INX	H	
0071	36	20			MVI	M.020H	010A	0D			DCRC		
0073	3A	4D	01	MOVI	LDA	014DH	010B	B9			CMPC		
0076	FE	00			CPI	0	010C	C8			RZ		
0078	CA	9E	00		JZ	UR	010D	C3	08	01	JMP	LOAD	
007B	FE	01			CPI	01H	0110	E5			REC	PUSH	H
007D	CA	A8	00		JZ	RT	0111	21	4D	01	LXI	H.014DH	
0080	FE	02			CPI	02H	0114	4E			MOV	C.M	
0082	CA	AF	00		JZ	DR	0115	77			MOV	M.A	
0085	FE	13			CPI	013H	0116	23			INX	H	
0087	CA	B9	00		JZ	DN	0117	71			MOV	M.C	
008A	FE	10			CPI	010H	0118	E1			POP	H	
008C	CA	C3	00		JZ	DL	0119	C9			RET		
008F	FE	11			CPI	011H	011A	E5			RND	PUSH	H
0091	CA	CD	00		JZ	LFT	011B	21	44	01	LXI	H.SH+3	
0094	FE	12			CPI	012H	011E	06	08		MVI	B.08H	
0096	CA	D4	00		JZ	UL	0120	7E			MOV	A.M	
0099	FE	03			CPI	03H	0121	07			RTOP	RLC	
009B	CA	DE	00		JZ	UP	0122	07			RLC		
009E	11	C2	FF	UR	LXI	D.00FFC2H	0123	07			RLC		

Listing 1: The Microbot program, assembled for an 8080 system which has the Processor Technology VDM-1 board located at hexadecimal locations CC00 to CFFF in memory address space (1024 bytes).

Listing 1, continued

```

0124 AE XRA M
0125 17 RAL
0126 17 RAL
0127 2D DCRL
0128 2D DCRL
0129 2D DCRL
012A 7E MOV A.M
012B 17 RAL
012C 77 MOV M.A
012D 2C INRL
012E 7E MOV A.M
012F 17 RAL
0130 77 MOV M.A
0131 2C INRL
0132 7E MOV A.M
0133 17 RAL
0134 77 MOV M.A
0135 2C INRL
0136 7E MOV A.M
0137 17 RAL
0138 77 MOV M.A
0139 05 DCRB
013A C2 21 01 JNZ RTOP
013D E6 03 ANI 03H
013F E1 POP H
0140 C9 RET
0141 SH DS 4
0145 AR DS 6
END

```

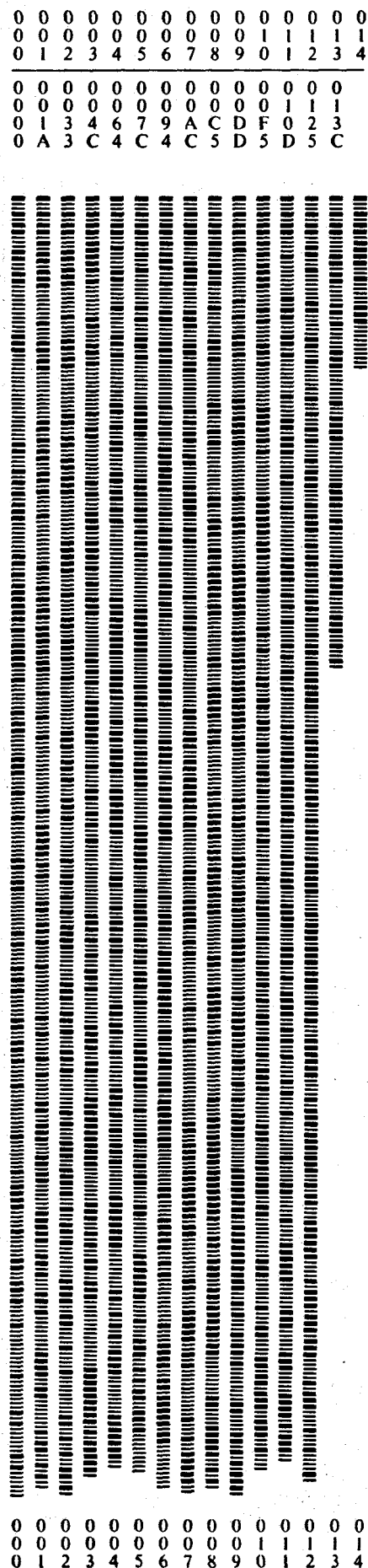
approaches. A measure of the average time it takes a microbot to fill all screen squares with his trail would provide indication of the efficiency of his move pattern. Various changes could be made to microbot's move routines and the results analyzed.

If Microbot's "room" of 16 x 64 squares proves too small for your needs, rooms of any size may be constructed and stored away in 1 K memory sectors. A check to sense that the creature was moving off the side of the screen could be used to call up another sector, move it to the screen and allow microbot to enter on the opposite side. Also, the same algorithm could be easily extended to displays with a higher resolution.

2. **Learning:** Some of the most complex and interesting programming work to be done is in the field of artificial intelligence and robot "learning." Microbot's use of direction status words should provide a capability for storing, retaining and modifying (eg: retracing steps) move paths that might be kept for future reference if useful, or discarded by the program if inappropriate or less efficient. Refer to texts on artificial intelligence for much more detailed discussions of what it means for a program to "learn."

Figure 3: The PAPERBYTES® bar code representation of the object code for the Microbot 8080 program by John Webster. The standard bar code frame format is used, with its synchronization byte (hexadecimal 96) followed by checksum byte, line identification byte, line length byte and data field. The data field of each frame uses the "absolute" format consisting of a 2 byte address followed by data to be stored at that address. The assembly of listing 1 was also typeset by machine from the same data file and contains the source code as well as the object code found in this bar code representation.

The documentation of bar code loader programs suitable for loading this program with the data in this figure is found in the book Bar Code Loader by Ken Budnick, available for \$2 at local computer stores and by mail from BITS Inc, POB 428, 25 route 101 West, Peterborough NH 03458. To read this data will require a homebrew or commercially manufactured bar code scanning wand, using procedures outlined in Ken Budnick's book.



3. **Sensing the Environment:** Simulation of optical, audio or tactile sensors would be very easy with microbots. Remote squares could be examined by microbots' "eyes," for example, according to any rules or limitations (such as distance and direction) the programmer wishes to impose. Experiment with obstacles of different shapes. Color or texture differentiation may be simulated by using other obstacle characters in addition to cursors, eliciting different responses from your creature. A starting point might be to have your microbot periodically scan one wall, searching for an "object" partially obscured by obstacles.

Other interesting possibilities would arise with two microbots in the same room leaving different trails and searching for each other. If you decide to explore this area, you might want to give toes to your microbots, so one microbot can tell which way the other's trail leads. Leaving a trail of three sequential numbers instead of asterisks might be one way.

4. **Work:** If you had a little robot running around your house, what would you like it to do? Vacuum the floor? Pick up small objects and put them away?

How about a robot that checks your potted plants and waters them if they need it? Or why not all these things at once? The possibilities are really limitless. Microbots will allow you to develop and test logical procedures for solving all of these problems.

5. **Feeding Time:** Don't forget that you are simulating a real working robot and that certain internal housekeeping routines should be built in.

Since any self-respecting real world robot will run on rechargeable batteries, a timer or move counter should be one of the first things built into your program to simulate "low batteries" and to initiate a search and docking routine for recharge at one or more "power stations" in the "room." Of course, you also have to remember to empty the vacuum cleaner or whatever.

Conclusion

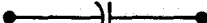
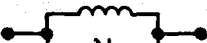
Sound interesting? Why not run Microbot and see what ideas occur to you? What robots really need right now is for lots of computer hobbyists to start thinking and working on their software. Oh yes, in case I forgot to mention it in my enthusiasm, it's also a lot of fun. ■

Linear Circuit Analysis

Leonard Anderson

Circuit analysis programs are valuable tools that can tell you how a circuit will work before you build it — "paper breadboards" in effect that don't require any component purchases, expensive equipment or debugging time spent on the bench. Analysis programs fall into two general categories: frequency domain and time domain. Presented here are the fundamentals of a frequency domain *linear* analysis program. In practice, linear analysis means that no active devices are operated at saturation or cutoff. Frequency domain tells us what circuits do at different frequencies, a type of analysis well-suited to model amplifiers, filters and operational amplifier circuits operating over any desired frequency range. You can make voltage and impedance readings at any point in the circuit without experiencing the loading problems that can occur with conventional equipment.

Because of the variations in small computer systems, primarily in memory, no specific language is given. All of the necessary flowcharts are presented along with necessary mathematical operations. You will need

Branch Type	Plus Node	Minus Node	Calculation of Admittance
Resistor			$Y = (1/R) + j0$
Capacitor			$Y = 0 + j(2\pi FC)$
Inductor			$Y = 0 - j[1/(2\pi FL)]$
Series RL			$Y = R/(R^2 + X^2) - j[X/(R^2 + X^2)]$ $X = 2\pi FL$
Series RC			$Y = R/(R^2 + X^2) + j[X/(R^2 + X^2)]$ $X = 1/(2\pi FC)$
Series LC			$Y = 0 + j[X/(1 - 2\pi FLX)]$ $X = 2\pi FC$
Parallel RL			$Y = (1/R) - j[1/(2\pi FL)]$
Parallel RC			$Y = (1/R) + j(2\pi FC)$
Parallel LC			$Y = 0 + j[(2\pi FC) - (1/2\pi FL)]$
Current generator			$Y = I + j0$
Current generator with θ			$Y = I \cos(\theta) + j[I \sin(\theta)]$

R = entered resistance in ohms.
L = entered inductance in henries.
C = entered capacitance in farads.
I = entered current in amperes.
F = solution frequency in hertz.

Editor's Note:

Readers wishing to obtain more details about the electrical concepts discussed in this article will find a concise treatment of complex impedance and related topics in *The Radio Amateur's Handbook* published by the American Radio Relay League, Newington CT 06111. For a good programmed learning introduction to electricity and DC circuits, see *Basic Electricity and DC Circuits* by Oliva and Dale, published by Texas Instruments Inc., POB 5012, Dallas TX 75222.

Table 1: Admittance calculations for simple branches. On both of the generators the direction of electron flow is shown by the arrow.

the four basic floating point functions plus array handling (single dimension arrays are acceptable, but two-dimensional arrays are preferred) and at least arctangent and logarithmic functions.

Matrix operations are done but you don't need BASIC MAT functions; the matrix is fully explained later. Using charts and explanations, you can write the program in any form from assembler to BASIC and higher languages. Assembler will work faster since this is a "number crunching" program. The

main constraints are memory and the ability to hold many arrays in main memory.

Modeling the Circuit

Each component of a circuit with two connections is called a *branch*. Connection points are called *nodes*. Several branches can be connected to the same node. Signal sources are also branches; these have specific requirements in node descriptions and are covered later.

Table 1 shows the basic branch types with only two nodes. The complex number *admittance* value is also given; the program is required to calculate the admittance as part of analysis.

Review of Complex Numbers

If you are unfamiliar with complex number notation or admittance, some review is necessary before beginning any programming. Complex numbers exist in either *rectangular* or *polar* form; rectangular form is used here. This requires two floating point numbers for every complex number. The lefthand number is called the *real* component and the righthand number, separated by the *j*, is called the *imaginary* component. Don't let the imaginary term fool you — it is very real. The naming comes from mathematical notation. [Note that electronics applications use *j* instead of the mathematical symbol *i* to avoid confusion with the symbol for current ... BWL]

Admittance is the reverse of impedance. You may be more familiar with impedance and the notation:

$$Z = R + jX$$

with *X* being *reactance*, either positive or negative. Admittance is:

$$Y = G + jB$$

with *G* being *conductance* and *B* *susceptance*, either positive or negative. The relationship $Y = 1/Z$ is true but there are special rules governing complex number mathematics. These rules are summarized in the text box on complex number arithmetic.

The circuit to be analyzed is converted into a model for the computer by copying each component as a branch. Each branch has two node numbers corresponding to connections in the circuit. All nodes above ground must have sequential numbering, beginning with 1, but the node numbers may be in arbitrary positions. A ground node is signified by 0.

A complete set of branch descriptions

Complex Number Arithmetic

The rules for rectangular form arithmetic are shown in the table below. Note that division and inversion have equal value denominators for both real and imaginary parts. Finding the denominator first in the division routine will increase computational speed. Note also that inverting $(C + jD)$ is equal to dividing $(A + jB)$ by $(C + jD)$ and setting *A* at unity and *B* at zero.

Actual test equipment such as oscilloscopes present magnitude and phase angle. This is the polar form, and conversion is as follows:

$$\begin{aligned} \text{Given rectangular form } A + jB, \\ \text{magnitude} = \text{MAGN} = \sqrt{A^2 + B^2} \text{ and} \\ \text{phase angle} = \text{PHA} = \arctangent (B/A). \end{aligned}$$

Conversion from polar to rectangular form is:

$$\begin{aligned} \text{Given polar form MAGN at angle PHA,} \\ \text{real part} = \text{MAGN} \times \cosine (\text{PHA}) \text{ and} \\ \text{imaginary part} = \text{MAGN} \times \sin (\text{PHA}). \end{aligned}$$

Trigonometric functions in the language will determine whether angles are in radians or degrees. Most are in radians. The conversion factor from radians to degrees is:

$$\text{Degrees} = 57.29577951 \times \text{radians}.$$

Given Complex Numbers of $(A + jB)$ and $(C + jD)$

Addition:

$$(A + jB) + (C + jD) = (A + C) + j(B + D)$$

Subtraction:

$$(A + jB) - (C + jD) = (A - C) + j(B - D)$$

Multiplication:

$$(A + jB) \times (C + jD) = (AC - BD) + j(AD + BC)$$

Division:

$$\frac{(A + jB)}{(C + jD)} = \left[\frac{AC + BD}{C^2 + D^2} \right] - j \left[\frac{AD - BC}{C^2 + D^2} \right]$$

Inversion:

$$\frac{1}{(C + jD)} = \left[\frac{C}{C^2 + D^2} \right] - j \left[\frac{D}{C^2 + D^2} \right]$$

Summary of complex arithmetic used for analysis.

comprises a circuit list. The circuit list we use requires three integer values and two floating point values to completely define a node. The three integer values are for the connections to other nodes and for indicating what number node we are presently at. The two floating point values define the complex admittance of the branch. To be useful, the minimum number of branches available should be at least 20.

You will notice that signal sources are currents and not voltages. This and use of admittances are deliberate in the solution of a node voltage. Consideration of node voltage solutions is important for our analysis.

Fundamental Circuit Matrix

A simple resistor network is shown in figure 1. This circuit can be analyzed quickly using pencil and paper, but will serve to show the mechanism of solutions. Keeping this circuit in mind, inspect the general matrix and simultaneous equations given in figure 2.

The mathematical matrix form and the simultaneous equations form are identical. Mechanizing the solution will require parts of both forms. Admittance subscripts will depend on node numbers (in the branch list) and follow certain rules depending on type. Those rules are given in table 2.

The simple example, expressed in the general equations of figure 2 is:

$$Y_{1,1} E_1 + Y_{1,2} E_2 = I_1 \quad (1)$$

$$Y_{2,1} E_1 + Y_{2,2} E_2 = I_2 \quad (2)$$

The conductance of 50 ohms is 0.02 mho, 25 ohms is 0.04 mho. [The mho is a unit of conductance, the inverse of the ohm. The mho is also called the siemens . . . CM] Using only the real parts of admittance and following the rules of table 2 yields the numeric forms:

$$0.06 E_1 - 0.04 E_2 = 1.0 \quad (1A)$$

$$-0.04 E_1 + 0.08 E_2 = 0 \quad (2A)$$

Solving for E_2 can be done by multiplying equation (1A) by 2/3 and adding the product to equation (2A) to give:

$$0.05333 E_2 = 0.66667$$

$$E_2 = 12.5$$

Substitution of E_2 into equation (1A) gives:

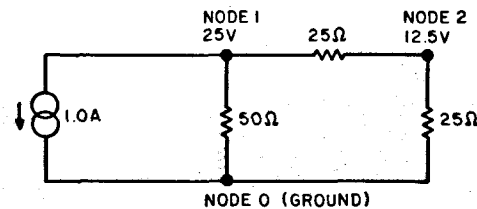


Figure 1: Simple resistive network with a current generator. The direction of the electron flow from the generator is indicated by an arrow. Note that nodes are numbered sequentially.

Matrix Representation									
$Y_{1,1}$	$Y_{1,2}$	$Y_{1,3}$	$Y_{1,4}$	...	$Y_{1,N}$	E_1		I_1	
$Y_{2,1}$	$Y_{2,2}$	$Y_{2,3}$	$Y_{2,4}$	...	$Y_{2,N}$	E_2		I_2	
$Y_{3,1}$	$Y_{3,2}$	$Y_{3,3}$	$Y_{3,4}$	...	$Y_{3,N}$	E_3	=	I_3	
$Y_{4,1}$	$Y_{4,2}$	$Y_{4,3}$	$Y_{4,4}$	...	$Y_{4,N}$	E_4		I_4	
.	.	.	.	...	.	.		.	
$Y_{N,1}$	$Y_{N,2}$	$Y_{N,3}$	$Y_{N,4}$	...	$Y_{N,N}$	E_N		I_N	
Simultaneous Equations									
$Y_{1,1}E_1 + Y_{1,2}E_2 + Y_{1,3}E_3 + Y_{1,4}E_4 + \dots Y_{1,N}E_N = I_1$									
$Y_{2,1}E_1 + Y_{2,2}E_2 + Y_{2,3}E_3 + Y_{2,4}E_4 + \dots Y_{2,N}E_N = I_2$									
$Y_{3,1}E_1 + Y_{3,2}E_2 + Y_{3,3}E_3 + Y_{3,4}E_4 + \dots Y_{3,N}E_N = I_3$									
$Y_{4,1}E_1 + Y_{4,2}E_2 + Y_{4,3}E_3 + Y_{4,4}E_4 + \dots Y_{4,N}E_N = I_4$									
$\vdots$						$\vdots$			
$Y_{N,1}E_1 + Y_{N,2}E_2 + Y_{N,3}E_3 + Y_{N,4}E_4 + \dots Y_{N,N}E_N = I_N$									

Figure 2: Matrix and simultaneous representations of any circuit. In the figure, and also in the text, the following conventions hold:

Y := circuit branch admittance

E := circuit node voltage

I := circuit node current

The circuit branch admittance is determined by the node description.

Matrix Array Subscripts			
Branch Type	Row	Column	Enter into Array by
Passive	Plus node	Plus node	Addition
	Plus node	Minus node	Subtraction
	Minus node	Minus node	Addition
	Minus node	Plus node	Subtraction
Generator	Plus node	$NMAX + 1$	Addition
	Minus node	$NMAX + 1$	Subtraction

Table 2: Matrix insertion subscript rules for branches. Any row and column node combination that contains a zero will not enter the matrix. NMAX is the maximum node number in the circuit. Remember that the nodes must be numbered sequentially.

$$0.06 E_1 - 0.50 = 1.0$$

$$E_1 = 1.50/0.06$$

$$E_1 = 25$$

The preceding straightforward mathematics becomes impractical for models with many nodes. Note that the subscript rules of table 2 are different than the example just given. A slightly different matrix arrangement is actually used.

Fundamental Properties of Matrices

Mechanization of the solutions requires a matrix in two dimensions having N rows and N+1 columns, N being the highest node number in the model. The first subscript is the row, and the second is the column position. The rightmost column is used only for signal sources.

Figure 3 shows the example represented by an array of two rows (maximum node number is 2) by three columns. Variable name M has been used for any matrix position. At the beginning of a solution all Ms are set to 0. The branch list is then inspected, values calculated and added or subtracted into the matrix array positions according to the rules of table 2.

The variable N is used to denote a numerator value. D denotes a denominator value of a multiplying factor. A circle indicates that the array variable is multiplied by the factor; an arrowhead indicates that the pro-

duct is subtracted from the array position. Only arrowhead marked positions are changed; all others are extracted and held in temporary storage.

Numeric values in figure 3 are the same as our simple example. The forward operation has these steps:

$$\text{Multiplication factor} = \left(\frac{M_{2,1}}{M_{1,1}} \right) = -0.66667.$$

$$\text{New } M_{2,2} = M_{2,2} - (\text{MULT.FACT} \times M_{1,2}) \\ = 0.05333$$

$$\text{New } M_{2,3} = M_{2,3} - (\text{MULT.FACT} \times M_{1,3}) \\ = 0.66667$$

Note that the steps are equivalent to finding E_2 by conventional mathematical operations. The *back* operating steps are the same as those for finding E_1 and are:

$$\text{Multiplication factor} = \left(\frac{M_{1,2}}{M_{2,2}} \right) = -0.75$$

$$\text{New } M_{1,3} = M_{1,3} - (\text{MULT.FACT} \times M_{2,3}) \\ = 1.50$$

Final node voltage solutions are indicated by numbered subscript Ns and Ds, where the subscript stands for the node of solution.

We have used a slightly different number handling scheme and have arrived at the same solution. This technique can be expanded to larger arrays in order to provide an algorithm that solves all node voltages. It should be noted that solutions provide node voltages with reference to circuit ground; modeling techniques allow finding the voltages between nodes.

A consideration for programming is array size. The array consumes $2 \times \text{NMAX} \times (\text{NMAX} + 1)$ floating point variables with NMAX referring to the maximum number of nodes allowed in any circuit. The actual physical size of the array is twice as large since both real and imaginary parts of the complex number must be stored. The example showed only real parts.

Final Matrix Solution Algorithm

A flowchart for the MATRIX solution algorithm is shown in figure 4. The routine assumes that all admittance values of a circuit have been calculated using the formulae in table 1 and have been entered into an initially 0 matrix at positions according to the rules of table 2. Figure 5 is a pictorial of a 4 node solution sequence using the same symbology as figure 3. The pictorial assumes

<table border="1"> <tr><td>M_{1,1}</td><td>M_{1,2}</td><td>M_{1,3}</td></tr> <tr><td>M_{2,1}</td><td>M_{2,2}</td><td>M_{2,3}</td></tr> </table>	M _{1,1}	M _{1,2}	M _{1,3}	M _{2,1}	M _{2,2}	M _{2,3}	INITIAL	<table border="1"> <tr><td>.06</td><td>-.04</td><td>1.0</td></tr> <tr><td>-.04</td><td>.08</td><td>0</td></tr> </table>	.06	-.04	1.0	-.04	.08	0
M _{1,1}	M _{1,2}	M _{1,3}												
M _{2,1}	M _{2,2}	M _{2,3}												
.06	-.04	1.0												
-.04	.08	0												
<table border="1"> <tr><td>D</td><td>0</td><td>0</td></tr> <tr><td>N</td><td>↓</td><td>↓</td></tr> </table>	D	0	0	N	↓	↓	FORWARD SOLUTION	<table border="1"> <tr><td>.06</td><td>-.04</td><td>1.0</td></tr> <tr><td>-.04</td><td>.05333</td><td>.66667</td></tr> </table>	.06	-.04	1.0	-.04	.05333	.66667
D	0	0												
N	↓	↓												
.06	-.04	1.0												
-.04	.05333	.66667												
<table border="1"> <tr><td></td><td>N</td><td>↑</td></tr> <tr><td></td><td>D</td><td>0</td></tr> </table>		N	↑		D	0	BACK SOLUTION	<table border="1"> <tr><td>.06</td><td>-.04</td><td>1.5</td></tr> <tr><td>-.04</td><td>.05333</td><td>.66667</td></tr> </table>	.06	-.04	1.5	-.04	.05333	.66667
	N	↑												
	D	0												
.06	-.04	1.5												
-.04	.05333	.66667												
<table border="1"> <tr><td>D₁</td><td></td><td>N₁</td></tr> <tr><td></td><td>D₂</td><td>N₂</td></tr> </table>	D ₁		N ₁		D ₂	N ₂	LOCATION OF SOLUTION NUMERIC VALUES	<table border="1"> <tr><td>.06</td><td></td><td>1.5</td></tr> <tr><td></td><td>.05333</td><td>.66667</td></tr> </table>	.06		1.5		.05333	.66667
D ₁		N ₁												
	D ₂	N ₂												
.06		1.5												
	.05333	.66667												

Figure 3: Graphical representation of a matrix solution to the simple example given in the text. Solutions are found at both of the nodes. The denominator of the solution is always found in the main diagonal, and the numerator of the solution is found in the generator column.

that the optional tests in figure 4 are not performed. This will give voltage solutions to all nodes.

IBM's venerable ECAP program yields all node voltages at each frequency. While useful, it can be difficult to interpret. A better way is to command the program for a specific node of solution (NS in the flowchart). The matrix must be solved at each frequency, so including the optional tests will reduce the number of back solutions to a minimum. A solution printout can then be made of all node voltage data at one node for the desired frequencies.

Calculation speed is increased by inverting the denominator in the outer loop. Division is invariably one of the most time-consuming functions; the inverted denominator allows multiplication instead of time-consuming division.

Another timesaving technique, not shown, is to include a 0 test of numerator $M(K,J)$ in the optional section. A 0 numerator will have no effect on inner loop variables, so a bypass could occur on 0s. (Remember to check both the real and imaginary parts of 0.) Many circuits will have only about one half array positions nonzero, so this test will help running time.

A similar test can be made by bypassing the inner loop if $M(J,I)$ is equal to 0. This helps reduce running time on large node circuit models; at 20 nodes or less, the help is arbitrary. All such tests take time, so it is worthwhile to perform these tests outside the inner loop since the inner loop iterates the most.

It is interesting to note where the node voltage solution numerators and denominators are located. Numerators are always in the righthand or generator column. Denominators are always in the main diagonal from upper left to lower right. Row position is equal to node of solution. Highest node numerators and denominators are always found; back solution is required only for nodes less than the highest node. The back solution algorithm is called the *Gaussian elimination* or *Gauss-Jordan* method of solution.

Complete Frequency Solution

The flowchart shown in figure 6 is the ANALYSIS routine. This routine assumes that the admittance of each branch, Y , is already calculated. Before the matrix branch values can be calculated, the entire matrix is set to 0. Zeroing the matrix will allow simple addition and subtraction in the matrix fill section.

Variable W is the frequency in radians per second. Variable $W1$ is the negative in-

verse of W . These simplify admittance calculations. An often used constant is 2π which should be stored as a single variable. Variable F is the solution frequency.

Variable Y is the calculated complex admittance for passive branches but is the stored value of current for generators. Variable M is the two-dimensional complex matrix array used in figure 3, and variable S is the solution matrix, which is capable of storing complex values. Variable L is the subscript value for array S .

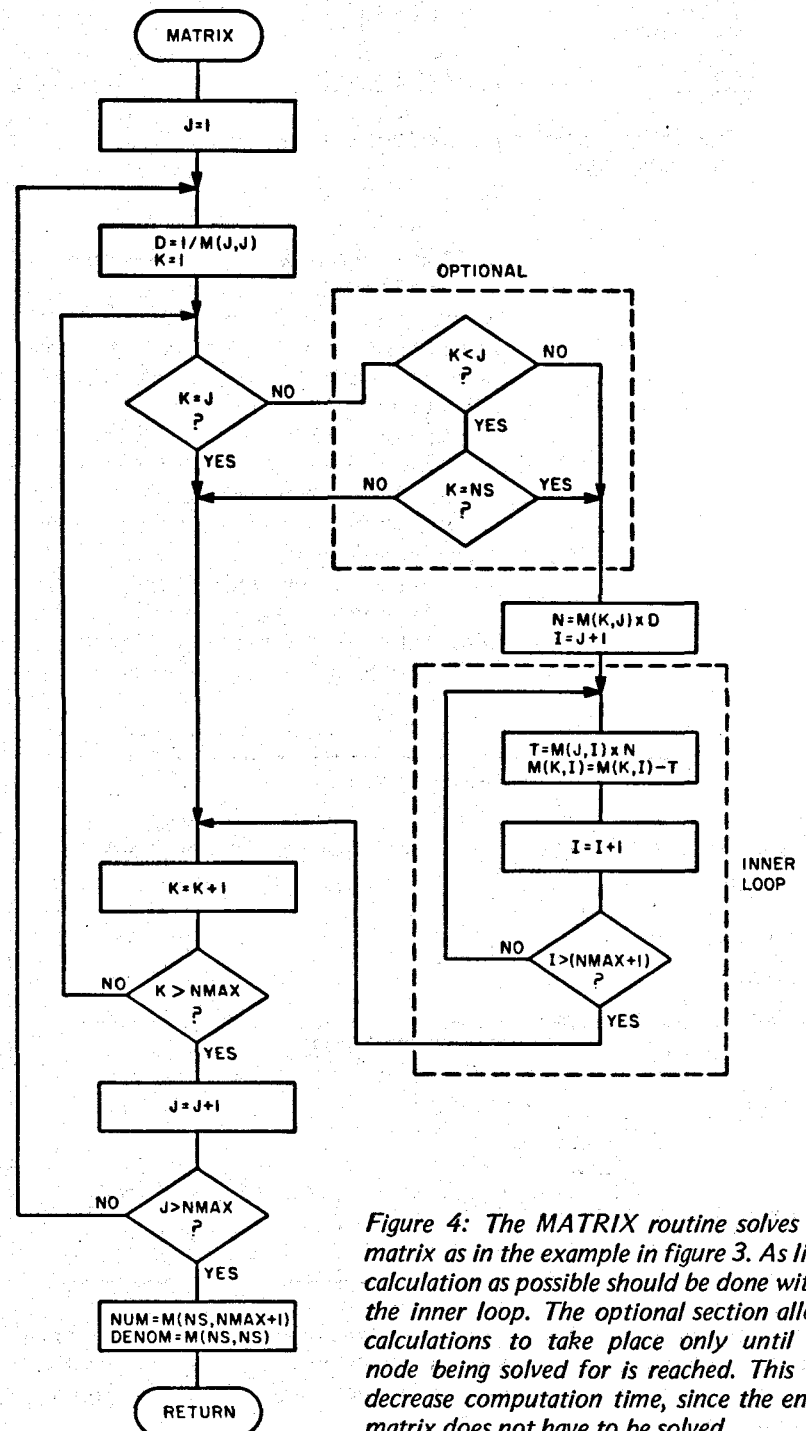


Figure 4: The MATRIX routine solves the matrix as in the example in figure 3. As little calculation as possible should be done within the inner loop. The optional section allows calculations to take place only until the node being solved for is reached. This will decrease computation time, since the entire matrix does not have to be solved.

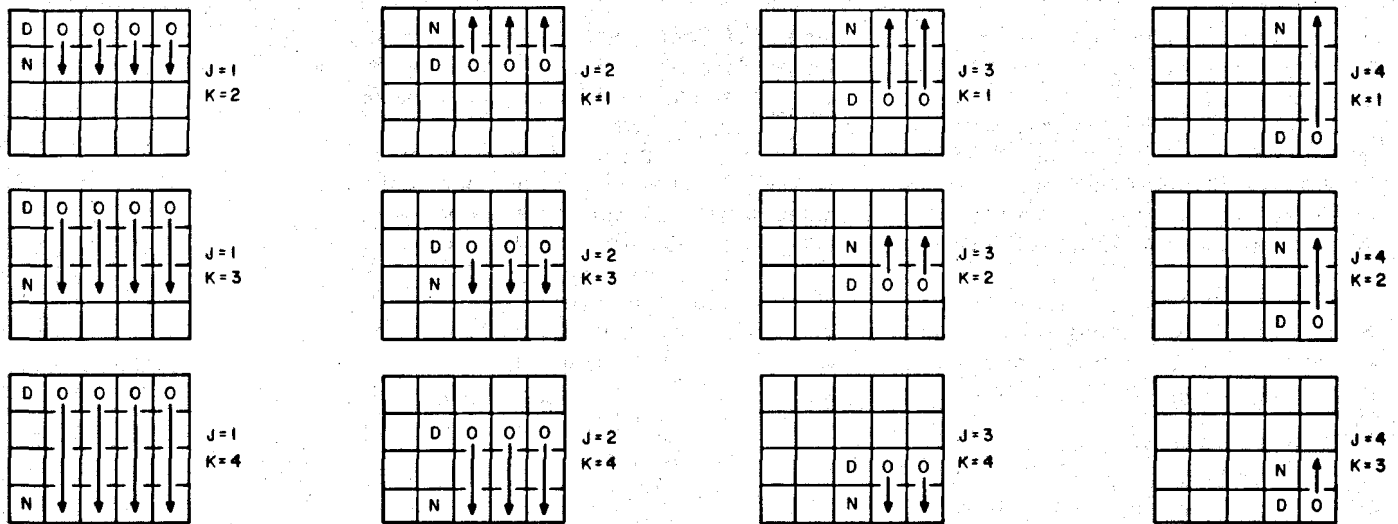


Figure 5: Sequential solution for all nodes of a 4 node circuit. This solution is the type that would take place if the optional section of figure 4 were deleted.

Most of the flowchart involves an examination of each branch, calculation of the admittance, and addition or subtraction of that value into the matrix. Positioning tests seem to be rather complex, but they do follow the rules of table 2. The flowchart and table 2 can be expanded to fit a special branch type.

Current flow of a generator branch is determined by node number entry order. This will be illustrated further under modeling techniques. A generator with one node set to zero will enter the matrix at only one position. If both nodes are nonzero the generator will enter the matrix at two positions.

Passive branches can have arbitrary node

ordering. The test flow allows for this. One node specified as 0 will cause calculated admittance to add at only one matrix position. If both nodes are nonzero, calculated admittance will add at two positions and subtract at two other positions.

To check the node test flow, go back to the example of figure 1 and the matrix values shown in figure 4. Remember that admittance, matrix and solution arrays of figures 4 and 6 require handling two floating point values per complex number. There is no way around this fact.

Branch Types and Passive Calculations

Small systems need simple rules, so it is probably easier to identify branch types in storage by integer numbers. One is sufficient. The number of different types should be considered in terms of calculation code and analysis needs.

Multicomponent branch types are strongly recommended since they reduce the number of nodes required in a model. Multicomponent branches are given in table 3 in terms of ohms, farads and henries stored in the branch list. List array storage positions should be considered in regard to the calculations.

Radian frequency, W , and its inverse negative, $W1$, are from the single frequency analysis routine. All of the parallel combinations are calculated as impedances first and then inverted. Series combinations should require less coding if calculated as impedances first; this can be seen by comparing table 3 with the complex values given in table 1.

Direct admittance calculations may be

Branch Value Storage		
Type	V1 (First Floating Point Variable)	V2 (Second Floating Point Variable)
Parallel RL	R	L
Parallel RC	R	C
Parallel LC	L	C
Series RL	R	L
Series RC	R	C
Series CL	C	L
Admittance Value Calculation		
Type	YR (Real Part)	YI (Imaginary Part)
Parallel RL	$1/V1$	$W1/V2$
Parallel RC	$1/V1$	$W \times V2$
Parallel LC	0	$W \times V2 + (W1/V1)$
Series RL	V1	$W \times V2$
Series RC	V1	$W1/V2$
Series CL	0	$W \times V2 + (W1/V1)$

Table 3: Calculations showing how a multicomponent branch admittance calculation is performed. The three series calculations (RL, RC, CL) are actually impedance calculations since they are so much easier to perform. To obtain the admittance, perform the complex inversion $Y = 1/(R+jX)$.

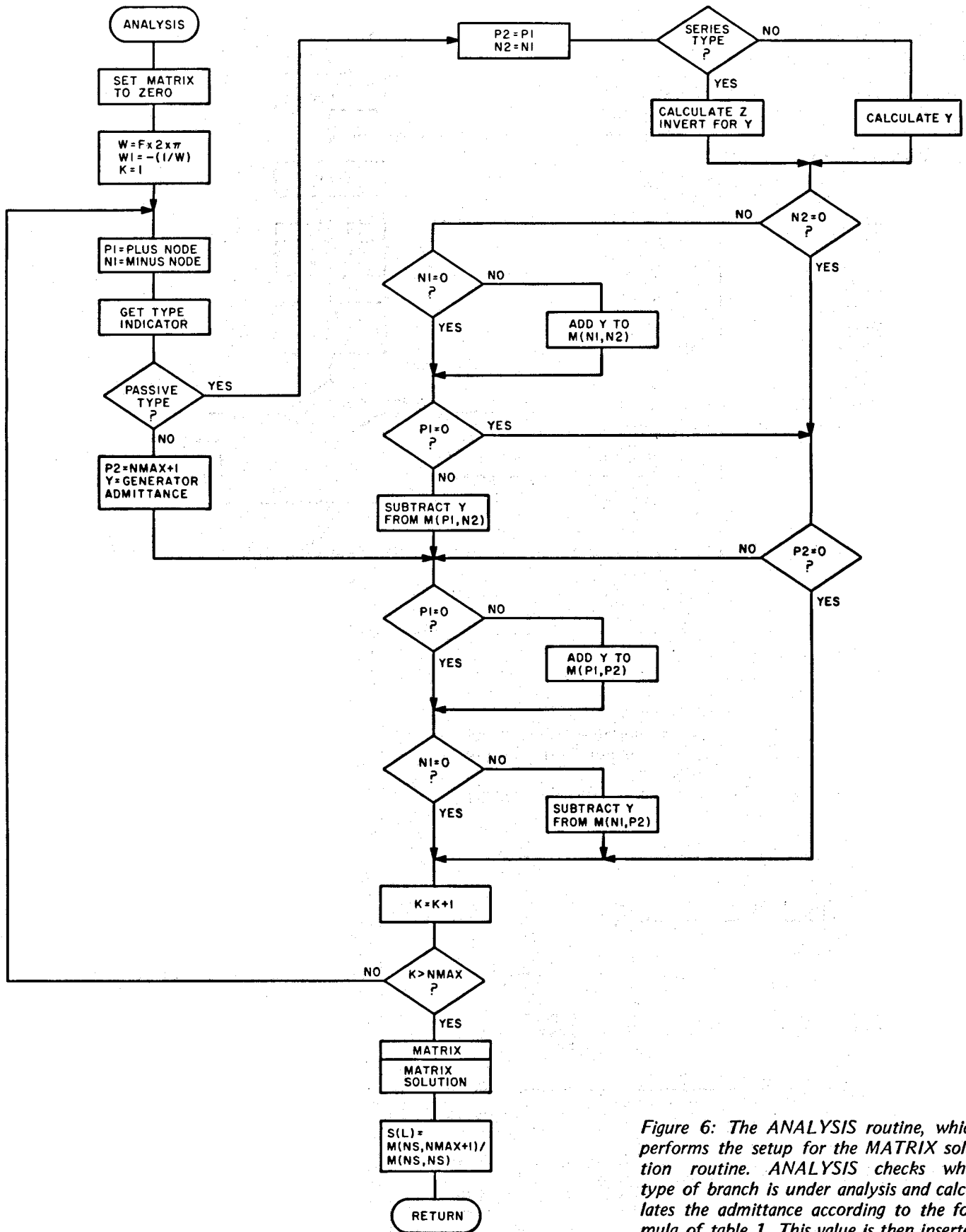
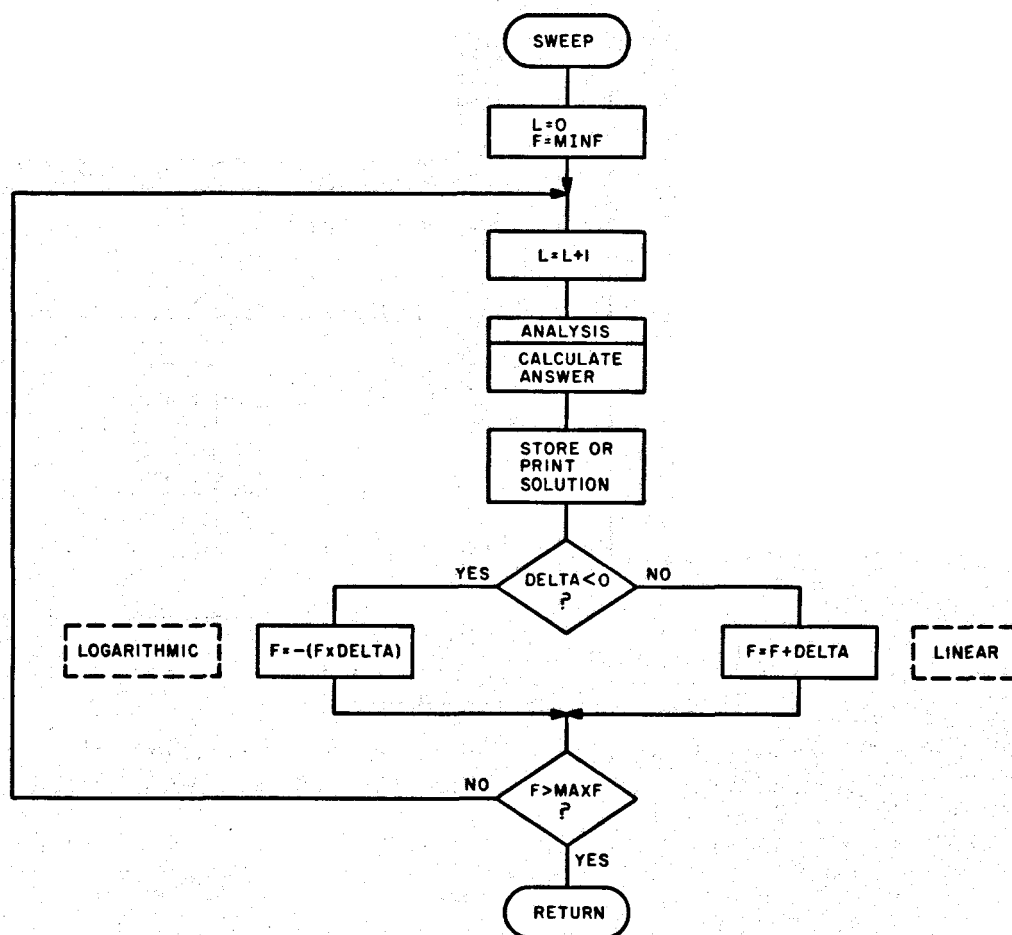


Figure 6: The ANALYSIS routine, which performs the setup for the MATRIX solution routine. ANALYSIS checks what type of branch is under analysis and calculates the admittance according to the formula of table 1. This value is then inserted into the matrix according to the rules of table 2. When all the nodes have been checked, MATRIX is called to perform the solution.

Figure 7: The SWEEP routine, used to perform a number of analyses on a particular circuit over a frequency range. Two types of sweeps are possible: linear and logarithmic. The variable DELTA is used as an increment and also as an indicator of which type of sweep analysis is to be performed. If DELTA is negative, a logarithmic analysis will be performed; if DELTA is positive, a linear analysis will occur.



slightly faster for series combinations. The choice is determined by the amount of memory, possibly by external memory control. All analysis matrices should be in main memory when ANALYSIS is called.

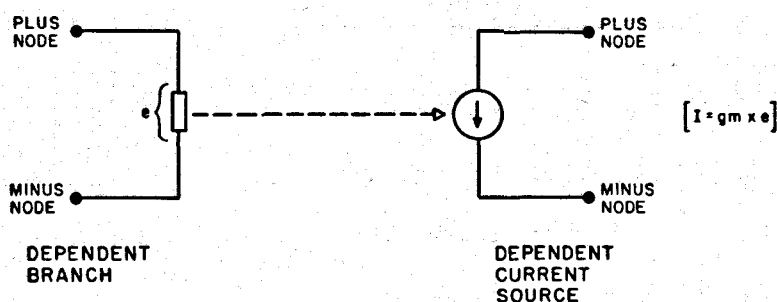
Passive Branch Values at DC

Direct current (DC) analysis can be considered as analysis at 0 Hz. Resistances remain the same but capacitors have 0 susceptance. Their susceptance (imaginary part) is effectively bypassed. Single inductors should have their susceptance value replaced by a low resistance, say a hundredth of an ohm (100 mhos susceptance), to avoid calculating difficulties. In actuality, inductors have finite resistance at DC.

Series combinations can be bypassed for certain types. A series resistor-capacitor (RC) or inductor-capacitor (LC) branch will have 0 admittance. In a series RL branch calculation, the susceptance calculation can be omitted and only the conductance calculation performed. Parallel combinations are also modified. A parallel RC branch requires only the conductance calculation. Parallel RL or LC branches would have the nominal 100 mhos conductance specified above. In all branches at 0 Hz the susceptance would be 0.

Frequency Sweeping

For this analysis, minimum and maximum analysis frequencies must be specified



Matrix Array Subscripts		
Row	Column	Enter into Array by
Plus node	Dependent branch minus node	Addition
Plus node	Dependent branch plus node	Subtraction
Minus node	Dependent branch plus node	Addition
Minus node	Dependent branch minus node	Subtraction

Table 4: Relationship of a dependent current source to it's branch. The source is entered into the matrix by the stated rules. Remember that any row and column combination with a zero node will not enter the matrix.

plus an increment. Most solutions over a narrow range will have a small increment but it is often useful to have a logarithmic frequency sweep with wide bandwidths. The main program should have a command point to which all major routines return. Frequency range can be selected at this command point with a minimum (MINF), a maximum (MAXF) and an increment (DELTA).

A choice between linear and logarithmic sweep can be done by simply checking for a negative or positive DELTA. A positive DELTA is the incremental change for a linear sweep, and a negative DELTA can be the frequency interval multiplier. The multiplier can be precalculated for the number of frequencies per decade:

$$\begin{aligned} \text{DELTA} &= -(\text{EXP}(2.302585/\text{NFD})) \\ &\quad \text{which reduces to} \\ \text{DELTA} &= -10^{(1/\text{NFD})} \end{aligned}$$

where NFD is the number of frequencies per decade and EXP is a function to raise e (the base for the natural logarithms which is approximately 2.71828) to the power of the following argument. For example, twenty frequencies per decade would have a DELTA equal to -1.122018.

Figure 7 is the flowchart for a frequency sweep analysis routine. The option of storing results or printing them directly depends on the operating system and memory. Subscript L is used only for solution matrix purposes.

There is a caution to be observed with solution storage: the number of frequencies to be analyzed must not exceed the storage matrix size. Other storage matrices may be written over if no check on the total number of frequencies analyzed is made. This is a very easy error to make.

Branch Switching

Implementing this function is useful for circuit modeling. In effect it allows you to disconnect or reconnect a branch from analysis and yet retain it in the branch list. It is the same as removing or replacing a component in a breadboard. If removed, it is still on the bench and can be installed later.

There are two easy ways to implement switching. Since we are using a numeric value to designate the branch type, we can define a positive value as a connected branch and a negative value as an open branch. Another method is to devote one byte per branch with a numeric value of +1 if connected and 0 or -1 if open. Another test can be inserted in the ANALYSIS routine of figure 6, just before the *passive type?* test. An open condition would bypass any calcu-

lations and go on to the next branch.

Dependent Current Sources

This branch type, not mentioned before, enables a model to duplicate transistors or operational amplifiers. It is a current source dependent on the voltage across another branch and is specified by a gain factor called *transconductance*. The symbology and matrix entry rules are given in table 4.

Transconductance is specified in mhos and is equal to the current divided by the voltage. Branch value entry is the transconductance, and admittance calculation for solution takes this as the stored value for the real part with an imaginary part of 0. You can think of transconductance as a *current gain* factor. A transconductance of 0.1 with a dependent branch voltage of 24 V produces a dependent source current of 2.4 A. A transconductance of 1.0 gives 24 A.

You do not have to be concerned about the dependent branch voltage. The matrix entry and solution will determine the current from the specified transconductance. The *direction* of electron flow is another factor and is determined by node number entry order of both the source and dependent branch. Reversing plus and minus node numbers of the source will reverse the source's flow; reversing node numbers of the dependent branch has the same effect; and reversing both source and dependent nodes returns electron flow to the original direction in the source.

One node of either source or dependent branch may be 0. Inspection of matrix entry subscript rules will show this. The only problem left is to allow the program to identify the dependent branch for subscripting.

Dependent Branch Identification

All passive components and generators may be entered into the branch list in any order. The ANALYSIS routine of figure 6 scans the entire list in order to fill the matrix for solution. A dependent source should be allowed to enter the list either ahead of or behind its dependent branch.

Two options come to mind. The extra integer byte mentioned under branch switching can be used to identify the dependent branch number in the list. Since transconductance is specified as the real part only, the second value storage might be used to hold the dependent branch number. Make sure that the particular floating point storage method that you are using will not change the value of the integer slightly. If there is any doubt about whether your floating

point to integer conversions are totally accurate, use the extra integer per branch for both switching and dependent branch identity. A dependent source can be switched by signing the extra integer. This method will be assumed for all further discussion.

Entering branch data in the list requires an extra entry for dependent sources to identify the dependent branch. The ability to print out a branch list should also include printing the dependent branch number.

Modification of the ANALYSIS routine to handle dependent current sources is shown in figure 8. This modification also

includes the switch function with the extra integer SW acting as the switch and identifier of dependent branch list number.

Impedance Analysis

All analysis procedures have so far given only voltage solutions. The last analysis we will consider is readily determined by disregarding all generators, placing a value of unity in the proper generator column position and then solving the matrix as before.

This may seem too simple. To understand it, consider once again the circuit of figure 1 and the position of solution numerators and denominators. Denominators always lie in the main diagonal and numerators always lie in the generator column. Im-

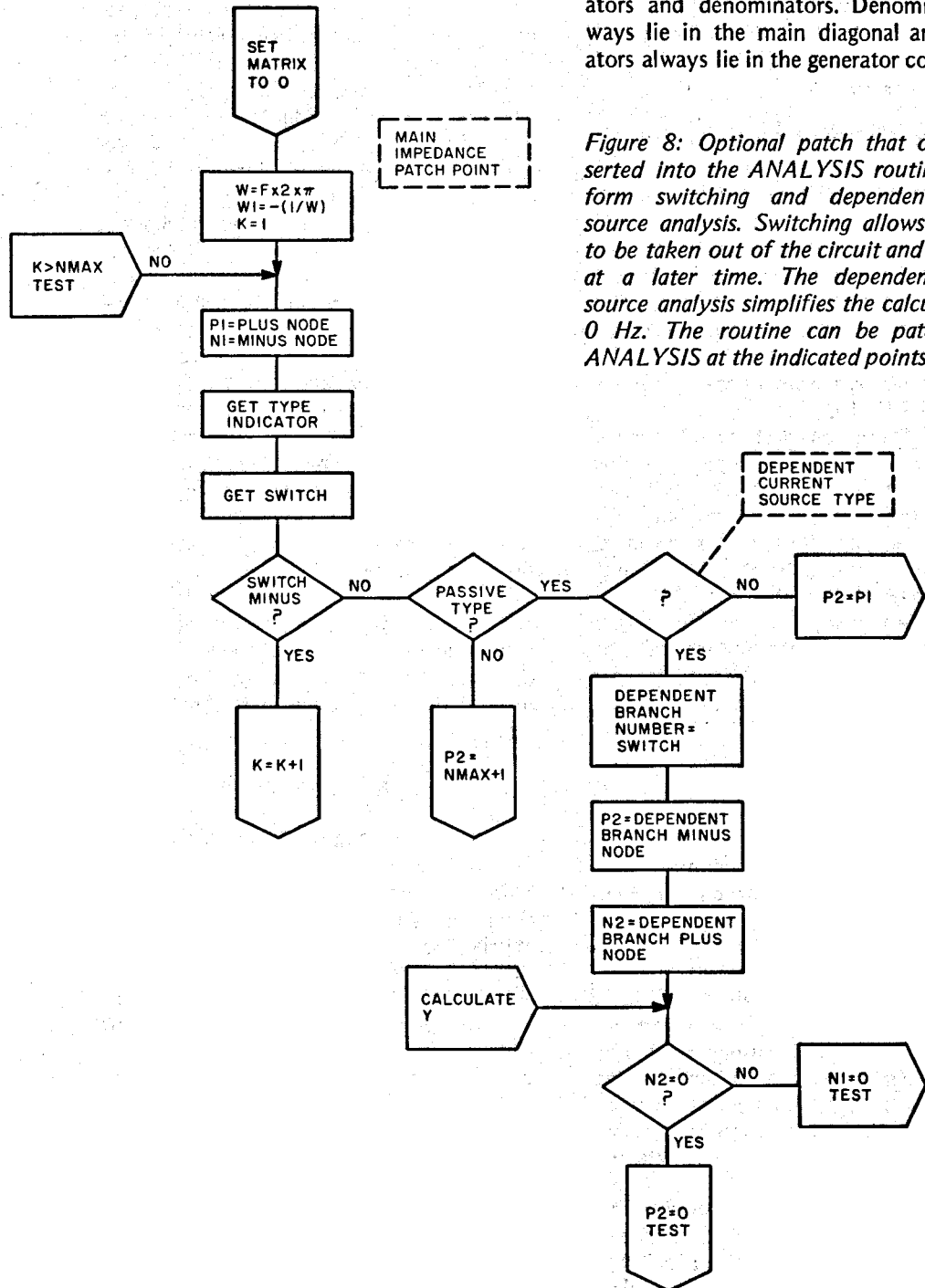


Figure 8: Optional patch that can be inserted into the ANALYSIS routine to perform switching and dependent current source analysis. Switching allows a branch to be taken out of the circuit and reinstated at a later time. The dependent current source analysis simplifies the calculations at 0 Hz. The routine can be patched into ANALYSIS at the indicated points.

pedance analysis will have only one generator column entry since all other generator branches automatically bypass any entry into the matrix.

A unity current is simply $(1.0 + j0)$, 1 A with no phase shift. This is the condition of figure 1 if an impedance is desired at node 1. The resistance of the total figure 1 circuit looking into node 1 is simply 25 ohms. In the model all generators are pure current sources; that is, they have no admittance themselves and can therefore be disregarded.

Addition of impedance analysis is shown in figure 9, a modification of figure 8. A flag variable must be held in the main program to identify analysis type. It is considered to be on for impedance and off for voltage analysis. It can be logical, integer or a single bit, but must be available if both types are desired.

Implementing impedance analysis requires a solution at only one node. It is best to use the optional tests in the flowchart of figure 4 and have all solutions at only one node. Direct printouts of impedance will be in rectangular form but the polar form can also be printed at the same time by using temporary variables and form conversion. Both forms are useful in studying analysis.■

REFERENCES

1. Huelsman, Lawrence P, *Basic Circuit Theory with Digital Computations*, Prentice-Hall, Englewood Cliffs NJ, 1972. Contains a great number of FORTRAN routines along with good basic circuit theory in both frequency and time domain.
2. Cornet, Wendell H Jr, and Battocletti, Frank E, *Electronic Circuits by System and Computer Analysis*, McGraw-Hill, New York, 1975. Again, FORTRAN oriented, but contains complete source code for Ohio State's linear and transient analysis programs.

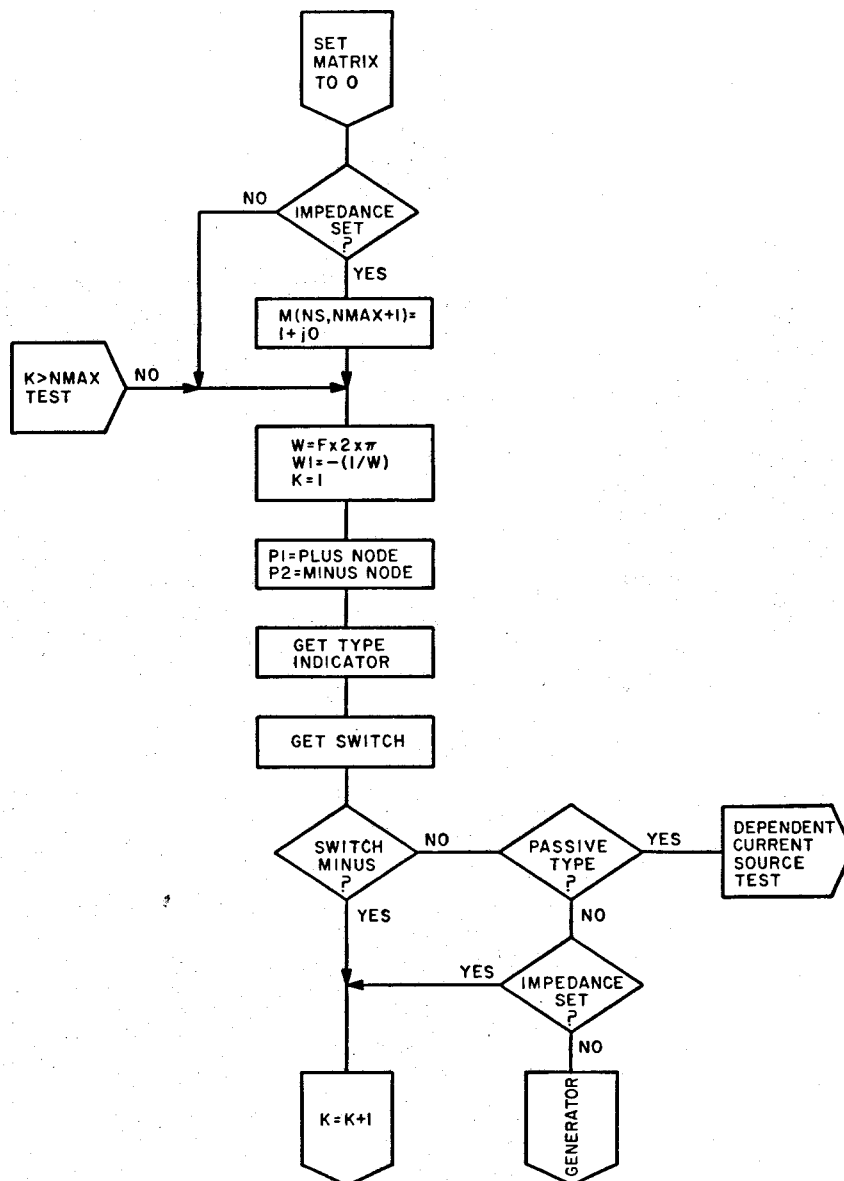


Figure 9: A patch that can be made to figure 8 to further increase the calculating power of the program. This patch will allow the solution of the matrix with impedance entries. It can be added to figure 8 at the instructions indicated.

Electronic Circuit Simulation

Robert Arp

This article is an introduction to applications which allow the design and study of electronic circuits with computer software instead of electronic hardware. Utilizing a computer to perform tedious mathematical calculations which represent circuit parameters is known as computer aided design (CAD). Computer aided design offers the following advantages to circuit designers:

- Components in an electronic circuit may be changed without fear of destroying any other circuit component.
- Results of component changes on circuit parameters may be quickly evaluated without breadboarding.
- Many complex phenomena may be simultaneously observed.
- Component evaluations may be accomplished before purchase.
- The behaviour of nonexistent components may be studied and their projected characteristics analyzed.

Methodology

Computer aided design may be discussed at three levels:

- General circuit simulations.
- Specific circuit simulations.
- Routine mathematical programs.

The first step towards utilizing a computer to design and analyze electronic circuits is the development of component representa-

tions which are palatable to the computer while being accurate enough to produce usable information for the designer.

Sophisticated packaged programs allow the user to describe the arrangement of components directly from the circuit, using special languages. These programs generate the necessary mathematical equations which must be solved to perform the desired analysis, then produce the specified data in a convenient format. In this article, we are primarily concerned with simulations that may be accomplished on personal computer systems with modest amounts of memory, along with hand held programmable calculators. The discussion will be restricted to topological representations that are compatible with languages readily available to personal computer owners and to languages supplied with programmable calculators.

We will transform circuit topology into sets of simultaneous equations. The set of equations describing a particular circuit may be manipulated to simulate changes in circuit parameters. Each change in a circuit parameter produces a slightly different set of equations which must be analyzed separately. This is where the speed of the computer is most noticeable. The program may also be prepared so that only the new data need be entered each time another solution is desired.

Passive circuit components will fit naturally into our sets of equations. Active circuit devices must be represented by models which are composed of passive com-

ponents. The circuit models presented here are identical to those usually prepared for the more sophisticated packaged programs. We will use the simplest model that provides the necessary accuracy. Because we wish to discuss general algorithms and analysis techniques rather than specific programs, we will use simplified circuit examples.

In order to utilize a computer efficiently, designers should be familiar with the following subjects:

- Methods of solving sets of simultaneous equations.
- Matrix algebra.
- Basic circuit theory.
- Device modeling.
- Parameter measuring methods.

Modeling Active Devices

Our algorithms will not accept circuit elements which are active, nonlinear, or time variant components. We will represent these elements with equivalent circuits which approximate the component being analyzed over a small region of interest. That is, we will simulate active, nonlinear, or time variant components with passive, linear, time invariant models. The particular model chosen for a circuit analysis is limited in accuracy primarily by the amount of work we wish to invest and available computer memory.

When constructing our models, we will focus our attention on those device parameters which will affect other circuit parameters and on those which will themselves be affected during the analysis. For example, the behaviour of a transistor depends not only on other circuit parameters and temperature. Its basic characteristics vary with its own behaviour in a particular circuit.

Our models will reflect the fact that the variations in these characteristics are small over particular regions of interest.

When we analyze the operation of a transistor, our attention is drawn to one or more of its normal regions of operation (cutoff, active, and saturation). Our model of a transistor must reflect the transistor's characteristics over these broad regions in which distinctive device parameters may be defined. In addition, our attention must be focused more finely so that it pinpoints the specific areas of interest in the cutoff, active, and saturation regions.

The size of our specific area of interest must be selected logically. When the device is to operate in all three regions, many small areas must be observed sequentially. Memory allocation and running time will both depend on the resolution of the model. Each model must be constructed with a resolution that is compatible with available memory, desired running time, and desired accuracy.

Piecewise Linearization

Piecewise linearization, as applied to the modeling of active semiconductor devices, is a method of approximating the nonlinear characteristic curves of these devices with sets of confluent straight line segments, as in figure 1. The accuracy of the model depends on the number of straight line segments used for the characteristic curve approximation. Model complexity increases rapidly as the number of straight line segments increase. Models which have the minimum resolution necessary are advantageous. Figure 1 shows a possible piecewise linear approximation (in the form of a dashed line) for the current versus voltage characteristic curve of a semiconductor diode.

Models of active devices can be composed of passive elements by using the piecewise linear approximations to the characteristic curves. The circuit to be analyzed is then redrawn with the models in place of the active devices. If the models are accurate enough, the operation of the new circuit will mimic that of the original.

The piecewise linear approximation for the silicon diode's characteristics shown in figure 1 is very crude. Therefore, in order to present the following discussion with the emphasis on segment construction, we will use a two-segment approximation.

Segment Construction

The characteristic curves of a device which must be available for model construction may be obtained from data sheets published by the manufacturer of the device,

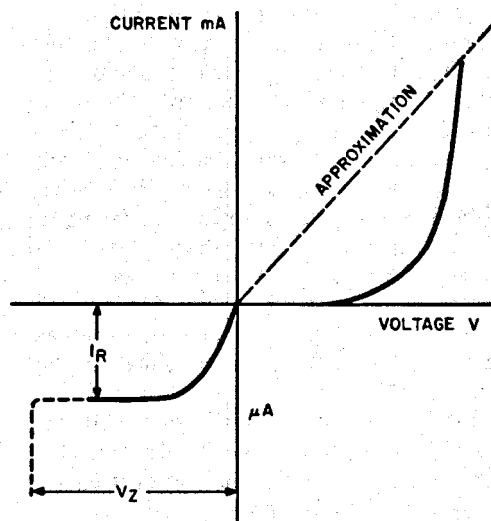


Figure 1: One segment of a piecewise linear approximation of a typical semiconductor diode current versus voltage characteristic curve. The approximation is shown with a single dashed segment.

or they may be prepared by the designer. The curves may also be plotted from measured data. However it is obtained, the characteristic curve for a silicon diode biased in the forward direction will be similar to that shown in figure 2. (Ignore the dotted lines for the moment.)

For each voltage value (V) between 0 V and 0.95 V which is applied directly across the diode, a current (I) will flow through the diode. The current which flows when the applied voltage is less than 0.6 V is very small. The current begins to increase significantly with increases in applied voltage only when the voltage is greater than 0.6 V. We call this point, 0.6 V, the threshold voltage (V_T). Synonyms for threshold voltage are offset, cutin, and break point voltage. The threshold voltage on diode characteristic curves is chosen arbitrarily. However, most designers agree that it should be considered to be between 0.5 V and 0.7 V for silicon diodes.

We could construct a simple two-segment piecewise linear approximation for the diode characteristic curve by placing a ruler over the curve and drawing one straight line from the origin through a point of the curve near the selected threshold value. A second straight line could then be drawn by placing the ruler, as closely as possible, parallel to the portion of the curve which is almost perpendicular to the voltage axis. The second segment would intersect the first segment at a point just to the right of the *knee* of the curve. The voltage coordinate at this point of intersection would be the threshold voltage of the piecewise linear approximation.

A more accurate and systematic technique for constructing the approximation to characteristic curves involves a fitting of the *best* line to a table of data consisting of voltage and current values obtained from the data sheet or experimental measurements. A program written to solve the equations in table 1 will provide the slope of each segment and the current which will flow through the diode model for a particular value of forward bias voltage applied to the diode.

Solutions to these equations provide a set of points through which linear segments that more closely fit the nonlinear curve may be drawn. In addition, the slope of each segment, which must be used in the schematic representation of the model, will be obtained. An algorithm level flowchart for the solution to these equations is shown in figure 3. The equations allow us to fit the *best* line to the data, instead of simply drawing the line by eye. This is especially important when the characteristic curves are prepared from experimental data.

The straight line segments that we will

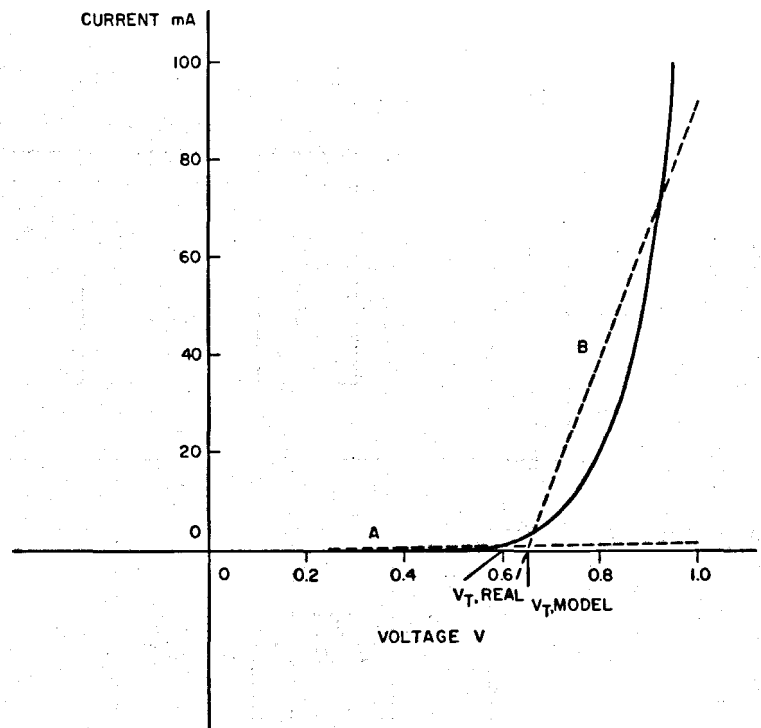


Figure 2: The forward current-versus-voltage characteristic of a silicon diode. Dotted lines represent the piecewise linear characteristic curve. $V_{T,real}$ is the actual threshold voltage of the diode. The threshold voltage moves to $V_{T,model}$ for the piecewise linear approximation.

use to represent a characteristic curve will intersect at certain coordinate points. These points of intersection are called *breakpoints*. The origin of a segment, when it does not begin at an intersection, is also a breakpoint. Likewise, the end of a segment, when it does not end at an intersection, is a breakpoint. In general, each straight line segment follows a portion of the characteristic curve rather closely, beginning at one breakpoint and ending at another. We may compute the proper current values at each of the breakpoints using the algorithm shown in figure 3.

$$\text{Slope} = m = \frac{\sum VI - \sum V \sum I / n}{\sum V^2 - (\sum V)^2 / n}$$

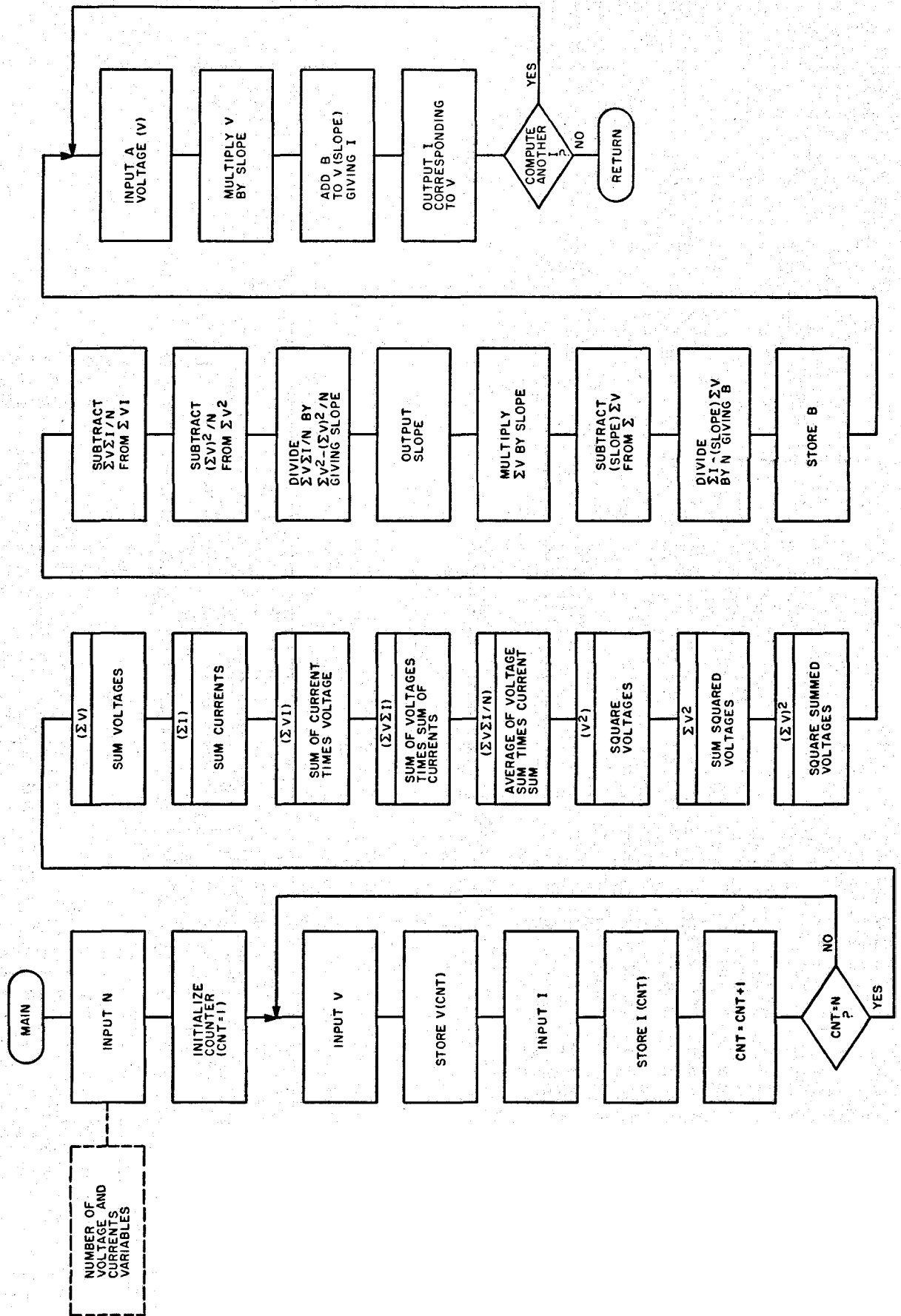
$$b = \frac{\sum I - m \sum V}{n}$$

$$I = mV + b$$

- b = value of I when V is 0.0 V
- I = forward bias current
- m = slope of a particular segment
- n = number of data points for a particular segment
- V = forward bias voltage

Table 1: Formula for fitting a line to a table of data. This method is known as finding the best line.

Figure 3: Method of least squares algorithm. The main program logic is shown here. The subroutines that are called are explained in the flowchart logic. Most of them are trivial programs. The program requires inputs that are the values of voltage, $V(\text{CNT})$, and current, $I(\text{CNT})$, which represent coordinate points on the current versus voltage characteristic for which linear segments are desired. The data must be entered and processed separately for each segment of the curve.



Having the means to construct accurate straight line approximations, we now turn our attention to the piecewise linear approximation of the silicon diode characteristic curve and a circuit model for the diode.

A Silicon Diode Model

The silicon diode model we construct must exhibit the following desirable characteristics:

- No current flows through the diode model when it is reverse biased.
- When the diode is forward biased, but the applied voltage is less than the threshold voltage, the amount of current flowing through the diode will increase very slightly as the applied voltage is increased.
- When the diode is forward biased, and the applied voltage is greater than the threshold voltage, the amount of current flowing through the diode will increase rapidly as the applied voltage is increased.

The data we will use to construct the characteristic curve approximation is that which was used to draw the curve itself. This data could be obtained by connecting a voltmeter across the leads of the diode and an ammeter in series with it while the applied voltage across the diode is increased from 0 V to 0.95 V. The tabulated data would appear as shown in table 2.

Our first segment, A, will be drawn from the coordinate point (0.0, 0.0) to the coordinate point (0.6, 0.001). Only the calculation which determines the slope of the segment is necessary because there are no intermediate points, and the characteristic curve is essentially linear between these two points. The remaining data pairs, including the breakpoint at (0.6, 0.001) are used to calculate the slope of segment B and two points through which the segment may be drawn. The results of the segment calculations are shown in table 3. The piecewise linear approximation is illustrated by the dotted lines in figure 2.

It should be obvious from both the piecewise linear approximation and table 3 that the behavior of the model will not precisely match that of the real diode. For example, table 3 shows that a forward bias of 0.95 V will cause a diode current of 80 mA to flow through the model. This voltage caused a current of 100 mA to flow through the real diode. Furthermore, the threshold voltage of the model, located at the intersection of segments A and B, is approximately 0.64 V — slightly higher than the threshold voltage of the real diode.

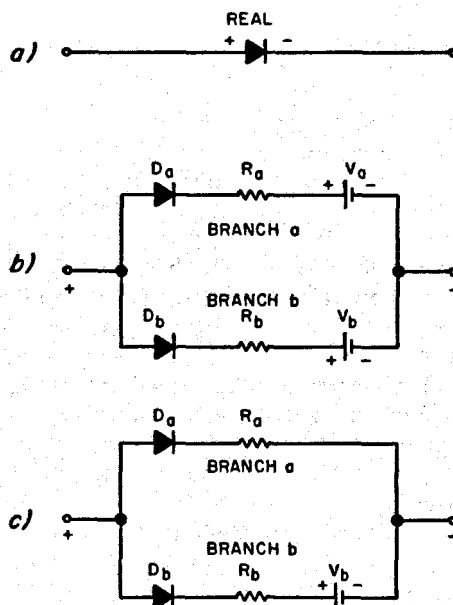


Figure 4: Two segment model for the silicon diode. Figure 4a is the schematic representation of the real diode. Figure 4b shows the model branches with ideal diodes. Figure 4c is the simplified model.

Each segment of the piecewise linear approximation to the characteristic curve may be represented schematically by a branch containing a resistor in series with an ideal diode and a battery whose voltage is equal to the breakpoint voltage at the beginning of the segment. The resistance in each branch depends on the slope of the segment which the branch represents and the slope of the previous branch. When the forward bias voltage across a branch exceeds the voltage of the battery in that branch, the ideal diode will be forward biased and the branch will behave as though the diode is not present. When a reverse bias voltage is applied across a branch, the ideal diode will act as an open circuit and no current will flow through the branch.

The branches with which we represent our diode model are shown in figure 4, along with the schematic representation for the real diode. Because segment A of the graph begins at 0 V, V_a in figure 4b is also 0 V. Therefore, we may eliminate the battery in branch a. The simplified model, with this element eliminated, is shown in figure 4c.

volts	amps
0.00	0.000
0.60	0.001
0.70	0.007
0.80	0.020
0.90	0.059
0.95	0.100

Table 2: Tabulated data for fitting the best straight line approximation to the silicon diode data shown in figure 2. Note that 0.6 is the threshold voltage (V_T).

segment	slope (m)	voltage (V)	current (A)	CC
A	0.0017	0.0	0.000	1.0
		0.6	0.001	
B	0.2667	0.70	0.013	0.9169
		0.95	0.080	

Table 3: Data resulting from the segment calculations showing slope and the starting and ending points on the graph. CC stands for the coefficient of correlation which indicates how good the fit is. A perfect fit will yield a coefficient of correlation of one.

$$\frac{1}{\frac{1}{R_a} + \frac{1}{R_b}} = \frac{1}{\text{slope B}}$$

$$\frac{1}{R_a} + \frac{1}{R_b} = \text{slope B}$$

$$\frac{1}{R_b} = \text{slope B} - \frac{1}{R_a}$$

$$\frac{1}{R_b} = \text{slope B} - \text{slope A}$$

$$\frac{1}{R_b} = 0.2667 - 0.0017$$

$$\frac{1}{R_b} = 0.265$$

$$R_b = 3.77 \text{ ohms}$$

Table 4: Calculation of the resistance in branch b in figure 4.

Figure 5: Diode model with element values shown. Diode D_a is forward biased whenever the input voltage (V_{IN}) is greater than zero. Diode D_b does not become forward biased until the input voltage is greater than 0.64 V.

The resistance of R_a is equal to the reciprocal of the slope of segment A, or:

$$R_a = \frac{1}{0.0017} = 588 \text{ ohms}$$

The slope of segment B is greater than the slope of segment A. To approximate segments A and B together, a resistor must be added to branch b such that the parallel combination of it and resistor R_a will equal the reciprocal of the slope of segment B whenever the voltage applied across the model is greater than V_b . The value of R_b is calculated in table 4. Note that R_b is equal to the reciprocal of the slope of segment A minus the slope of segment B.

The current that will flow through the diode model for any applied voltage may be

calculated by analyzing the circuit shown in figure 5. Ideal diode D_a is forward biased and acts as a short circuit whenever the input voltage (V_{IN}) is greater than 0 V. For input voltages less than V_b , ideal diode D_b acts as an open circuit and the circuit equation becomes:

$$I = \frac{V_{IN}}{R_a}$$

However, for input voltages greater than V_b , ideal diode D_b is forward-biased and the circuit acts as though diodes D_b and D_a are short circuits. For this case, a single node equation may be written for the circuit of figure 5:

$$\frac{V_{IN} - V_b}{R_b} + \frac{V_{IN}}{R_a} = 0$$

$$\frac{V_{IN}}{R_a} = I_a, \frac{V_{IN} - V_b}{R_b} = I_b$$

$$I = I_a + I_b$$

For example, if V_{IN} is set to 0.8 V:

$$I_a = \frac{0.8}{588} = 1.36 \text{ mA}$$

$$I_b = \frac{0.8 - 0.64}{3.77} = 42.4 \text{ mA}$$

$$I = 1.36 + 42.4 = 43.7 \text{ mA}$$

A designer may use the piecewise linear approximation to the characteristic curve or sample calculations as illustrated above to

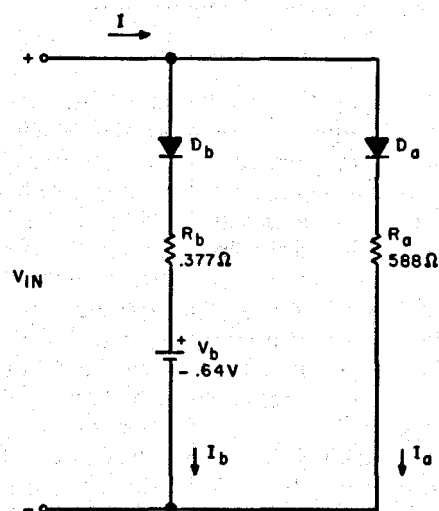
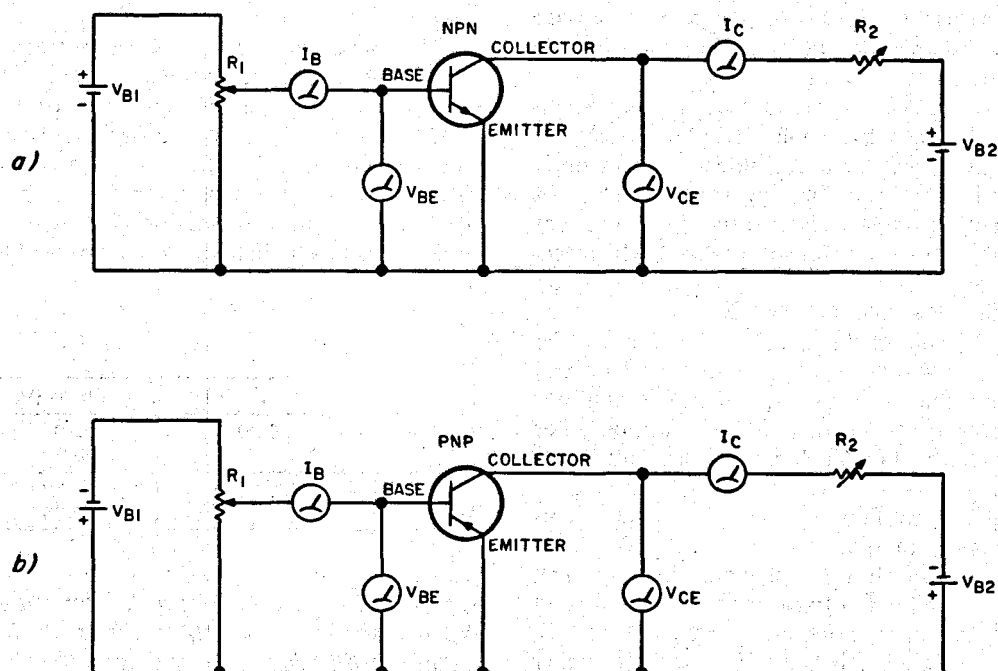


Figure 6: Meter connections for measuring transistor parameters: Figure 6a is for an NPN transistor, and 6b shows the connections for a PNP transistor.



determine the suitability of the model. If a more accurate model is necessary, another segment may be added to the piecewise linear approximation to obtain greater resolution.

The techniques used to obtain a model for the silicon diode are also applicable to transistor modeling. The transistor model is more complicated, and requires much more data. We proceed now to an example using a 2N5192 transistor wired in a common emitter configuration. The data to be used was obtained experimentally.

An NPN Transistor Model

The data which must be available to construct a transistor model depends upon the configuration in which the transistor is used. The following information must be included in a complete model of a transistor used in a common emitter configuration:

- h_{ie} : = common emitter input impedance.
- h_{oe} : = common emitter output admittance.
- β : = the DC forward current gain.
- R_{sat} : = collector-to-emitter saturation resistance.
- V_T : = base-to-emitter threshold voltage.

Figure 6a shows a measuring circuit which may be used to obtain the necessary common emitter data for an NPN transistor. Figure 6b shows a similar circuit which may be used with PNP transistors. When using the measurement circuits of figure 6, a FETVOM or similar high input impedance multimeter must be used to measure the voltage between the base and emitter (V_{BE}). In addition, the transistor should be mounted to a large heatsink to prevent erratic meter readings due to changes in device temperature.

The equations in table 5 apply to the measuring circuits and the characteristic curves which are plotted using the data obtained with them. The equations describe which measurement circuit parameters must be held constant in each equation while those in the denominator are varied to obtain the data expressed by the numerator.

The variables h_{oe} , β , and R_{sat} are obtained from the collector current versus collector-to-emitter voltage drop characteristics for the transistor after they are plotted with the data obtained with the measurement circuit. The variables h_{ie} and V_T are obtained from the base current versus the base-to-emitter voltage drop curves.

Figure 7 shows the common emitter input characteristics for the transistor we

$h_{oe} = \frac{\Delta I_C}{\Delta V_{CE}} \bigg _{I_B}$	I_B kept constant with dimensions (A/V)
$h_{ie} = \frac{\Delta V_{BE}}{\Delta I_B} \bigg _{V_{CE}}$	V_{CE} kept constant with dimensions (ohms)
$h_{fe} = \beta = \frac{\Delta I_C}{\Delta I_B} \bigg _{V_{CE}}$	V_{CE} kept constant without dimensions
$R_{sat} = \frac{\Delta V_{CE,sat}}{\Delta I_{C,sat}} \bigg _{I_B}$	I_B kept constant with dimensions (ohms)
V_T	= threshold voltage

Table 5: These equations apply to the measuring circuits shown in figure 6. They calculate the common emitter output admittance (h_{oe}), the common emitter input impedance (h_{ie}), the Dc forward current gain (β), the collector-to-emitter saturation resistance (R_{sat}), and the base-to-emitter threshold voltage (V_T).

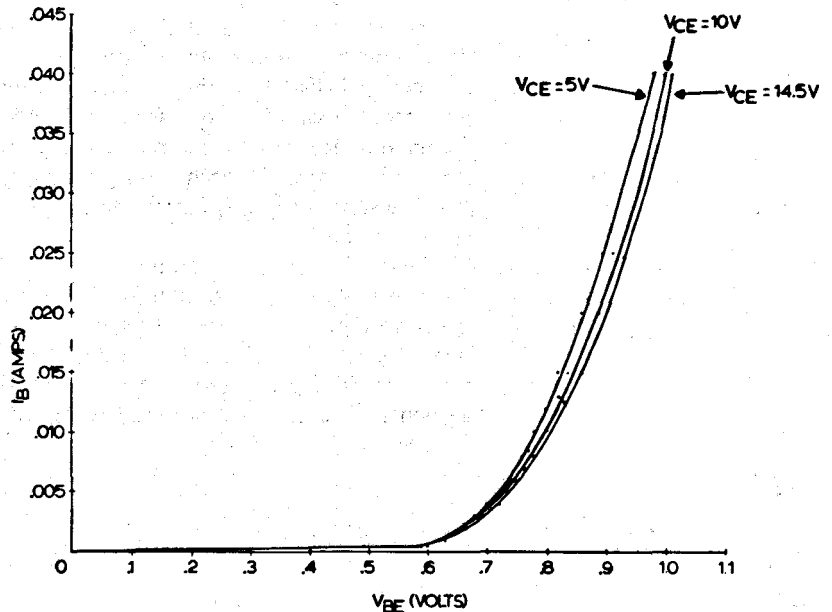


Figure 7: Graph of the nonlinear common emitter input characteristics for a 2N5192 transistor.

will model. In figure 7, base current (I_B) versus base-to-emitter voltage drop (V_{BE}) is plotted for three values of collector-to-emitter voltage drop (V_{CE}). This figure illustrates the very small shift in the input characteristics when the collector-to-emitter voltage is varied. Furthermore, the divergence of the curves is very slight. For these reasons, and also ease of calculation, the collector-to-emitter voltage drop curve of 10 V will be chosen as representative of the family of input curves.

Segment	Measured Data		Slope	Graphing Coordinates	
	V_{BE} (volts)	I_B (amps)		V_{BE} (volts)	I_B (amps)
A	0.00	0.000	0.001246	0.00	≈ 0
	0.59	0.0005		0.63	0.0008
	0.63	0.001			
B	0.63	0.001	0.042713	0.63	0.0008
	0.66	0.002		0.72	0.0047
	0.682	0.003			
	0.705	0.004			
	0.72	0.005			
C	0.72	0.005	0.086346	0.72	0.0047
	0.74	0.006		0.835	0.0144
	0.77	0.0085			
	0.78	0.010			
	0.82	0.013			
	0.835	0.015			
D	0.835	0.015	0.152862	0.835	0.0144
	0.875	0.020		1.00	0.0395
	0.91	0.025			
	1.00	0.040			

Table 6: Data used to compute the slope of each piecewise linear segment of figure 7 and the current at the segment data breakpoints.

The data used in obtaining the piecewise linear approximation of the input characteristic curves is listed in table 6 along with the associated approximation segments. Also shown in table 6 are the computed values for the base current at each data breakpoint (not the segment breakpoints) and the slope of each segment.

The data breakpoint graphing coordinates listed in table 6 are plotted in figure 8. They are connected by the four straight line segments A, B, C and D. Other coordinate points could have been used to plot the segments. The data breakpoints were arbitrary.

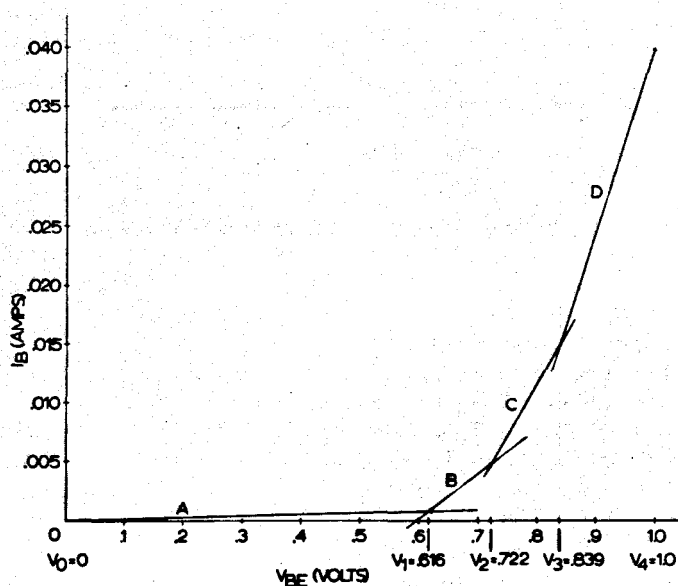


Figure 8: Accurate piecewise linear approximation to the input characteristics shown in figure 7.

trarily chosen.

In figure 8 it can be seen that segment A intersects the origin at V_{BE} equal to 0 V and segment B at V_{BE} equal to 0.616 V. Segment B, in addition to the segment A intersection, intersects segment C at V_{BE} equal to 0.722 V. Segment C, in addition to the segment B intersection, intersects segment D at V_{BE} equal to 0.839 V. Segment D discontinues at V_{BE} equal to 1.0 V. We will use the first four base-to-emitter voltage drop values as the breakpoints of the base-emitter representation in the transistor model.

As in the case of the silicon diode, each segment of the piecewise linear approximation to the input characteristic curve may be represented as a resistor in series with an ideal diode and a battery whose voltage is equal to the breakpoint voltage at the beginning of the segment. Often the first segment of a piecewise linear approximation need not be included in the model because its effect on a circuit is minimal. Assuming this condition, the base-emitter junction would be considered off until the base-to-emitter voltage drop is greater than the first breakpoint voltage past the origin.

If we omit segment A of the piecewise linear approximation, segment B should begin at its point of intersection with the V_{BE} axis, where V_{BE} equals 0.595 V and I_B equals 0 A. However, we may arbitrarily choose to let the threshold voltage for the model remain at 0.616 V. We may wish to do this in order to improve the accuracy of the model. The schematic representation of segment B is shown in figure 9. The branch resistance, R_b , is equal to the reciprocal of the slope of segment B. The voltage of the

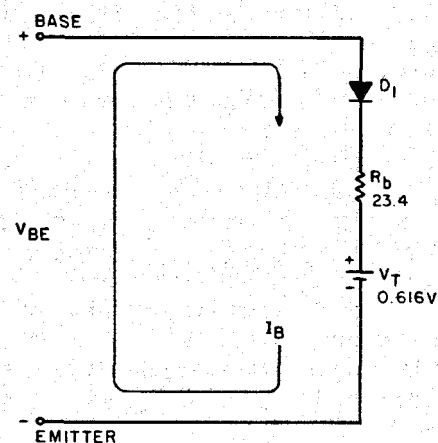


Figure 9: Schematic representation for segment B of the common emitter input characteristics piecewise linear approximation for the 2N5192 transistor. The resistance of resistor R_b equals the inverse of the slope of segment B. Values of the slope of B are listed in table 6.

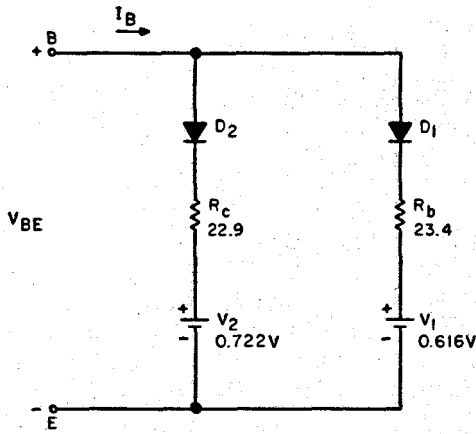


Figure 10: Partial model of the common emitter input characteristics for the 2N5192 transistor.

battery, V_T , is equal to the threshold voltage of the model. As in the diode model, when the base-to-emitter voltage is greater than the threshold voltage, the branch will behave as though the ideal diode is a short circuit.

We complete the model for the transistor input characteristics by expanding the circuit of figure 9 with representations for the other segments as shown in figure 10. The slope of segment C is greater than the slope of segment B (input impedance, h_{ie} , decreases when the base-to-emitter voltage drop is greater than V_2). When the segment C branch is added to the segment B branch, the resistor in the segment C branch must have a value of resistance which combines with the resistance of R_b to yield a total parallel resistance that is equal to the reciprocal of the slope of segment C whenever the base-to-emitter voltage drop is greater than V_2 . The value of R_c is:

$$\frac{1}{\frac{1}{R_b} + \frac{1}{R_c}} = \frac{1}{\text{slope C}}$$

$$\frac{1}{R_b} + \frac{1}{R_c} = \text{slope C}$$

$$\frac{1}{R_c} = \text{slope C} - \frac{1}{R_b}$$

$$R_c = \frac{1}{\text{slope C} - \frac{1}{R_b}}$$

$$R_c = \frac{1}{0.086346 - 0.042735}$$

$$R_c = 22.9 \text{ ohms}$$

A more accurate way to calculate R_c would have been to use the slope of segment B instead of the reciprocal of the resistance R_b .

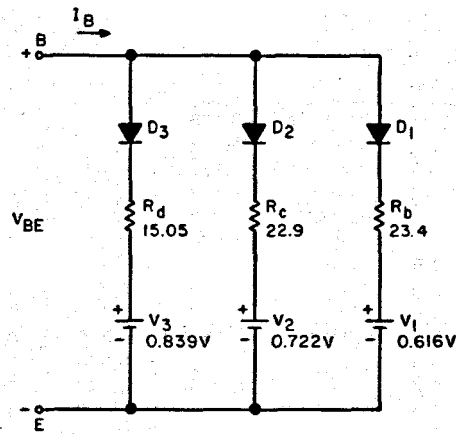


Figure 11: Complete base-emitter model for the 2N5192 wired in a common emitter configuration.

	$V_{BE} < V_1$	$D_1 \text{ off}, D_2 \text{ off}$	(branches b and c are off)
$V_1 < V_{BE} < V_2$	$V_{BE} < V_2$	$D_1 \text{ on}, D_2 \text{ off}$	(branch b on, branch c off)
$V_{BE} > V_2$	$V_{BE} > V_2$	$D_1 \text{ on}, D_2 \text{ on}$	(branch b on, branch c on)

Table 7: These inequalities specify the condition of the branches in the partial model of figure 10.

	$V_{BE} < V_1$	$D_1 \text{ off}, D_2 \text{ off}, D_3 \text{ off}$	(all branches off)
$V_1 < V_{BE} < V_2$	$V_{BE} < V_2$	$D_1 \text{ on}, D_2 \text{ off}, D_3 \text{ off}$	(branch b on, branches c and d off)
$V_2 < V_{BE} < V_3$	$V_{BE} < V_3$	$D_1 \text{ on}, D_2 \text{ on}, D_3 \text{ off}$	(branches b and c on, branch d off)
$V_{BE} > V_3$	$V_{BE} > V_3$	$D_1 \text{ on}, D_2 \text{ on}, D_3 \text{ on}$	(all branches on)

Table 8: These inequalities specify the condition of the branches when the base to emitter voltage is applied to the model in figure 11.

The circuit shown in figure 10 illustrates the addition of branch c to branch b. The inequalities in table 7 specify the condition of the branches in the partial model of figure 10.

Segment D is added to the circuit of figure 10 in the same manner that segment C was added to segment B. The value of R_d is:

$$\frac{1}{\frac{1}{R_b} + \frac{1}{R_c} + \frac{1}{R_d}} = \frac{1}{\text{slope D}}$$

$$\frac{1}{R_b} + \frac{1}{R_c} + \frac{1}{R_d} = \text{slope D}$$

$$\frac{1}{R_d} = \text{slope D} - \frac{1}{R_b} - \frac{1}{R_c}$$

$$R_d = \frac{1}{\text{slope D} - \frac{1}{R_b} - \frac{1}{R_c}}$$

$$R_d = \frac{1}{0.152862 - 0.042735 - 0.043668}$$

$$R_d = 15.05 \text{ ohms}$$

Table 9: Common emitter output data for the 2N5192 transistor. The base current (I_B) and collector current (I_C) are given in amperes. This data was obtained with the measurement circuit of figure 6a.

I_B	0.001	0.003	0.006	0.009	0.012	0.015	0.018	0.021	0.024
V_{CE}	I_C	I_C	I_C	I_C	I_C	I_C	I_C	I_C	I_C
0.5	0.0457	0.175	—	—	—	—	—	—	—
0.6	0.0458	0.176	—	—	—	—	—	—	—
0.7	0.0459	0.177	—	—	—	—	—	—	—
0.8	—	0.178	—	—	—	—	—	—	—
0.9	0.04595	0.178	—	—	—	—	—	—	—
1.0	0.046	0.178	0.339	—	—	—	—	—	—
1.5	—	—	0.340	—	—	—	—	—	—
2.0	—	0.179	0.341	0.520	—	—	—	—	—
2.5	—	—	0.345	0.521	—	—	—	—	—
3.0	0.0461	0.1795	0.346	0.530	0.660	—	—	—	—
3.5	—	—	0.350	0.535	0.670	0.802	—	—	—
4.0	—	0.180	0.351	0.540	0.675	0.815	0.940	—	—
4.5	—	—	0.359	0.543	0.680	0.825	0.959	1.065	—
5.0	0.0462	—	0.360	0.550	0.690	0.835	0.965	1.080	1.195
6.0	—	0.181	0.365	0.560	0.700	0.850	0.985	1.10	1.220
7.0	—	—	0.370	0.565	0.720	0.870	1.005	1.120	1.245
8.0	—	0.183	0.379	0.580	0.730	0.885	1.025	1.140	1.265
9.0	—	—	0.381	0.583	0.740	0.905	1.040	1.160	1.285
10.0	0.048	0.190	0.395	0.595	0.755	0.920	1.060	1.180	1.300
11.0	—	—	0.400	0.605	0.770	0.940	1.075	1.195	1.320
12.0	—	0.195	0.405	0.619	0.790	0.955	1.100	1.225	1.350
13.0	—	—	0.415	0.638	0.800	0.980	1.120	1.245	1.375
14.0	0.049	0.200	0.420	0.650	0.820	1.0	1.140	1.260	1.395

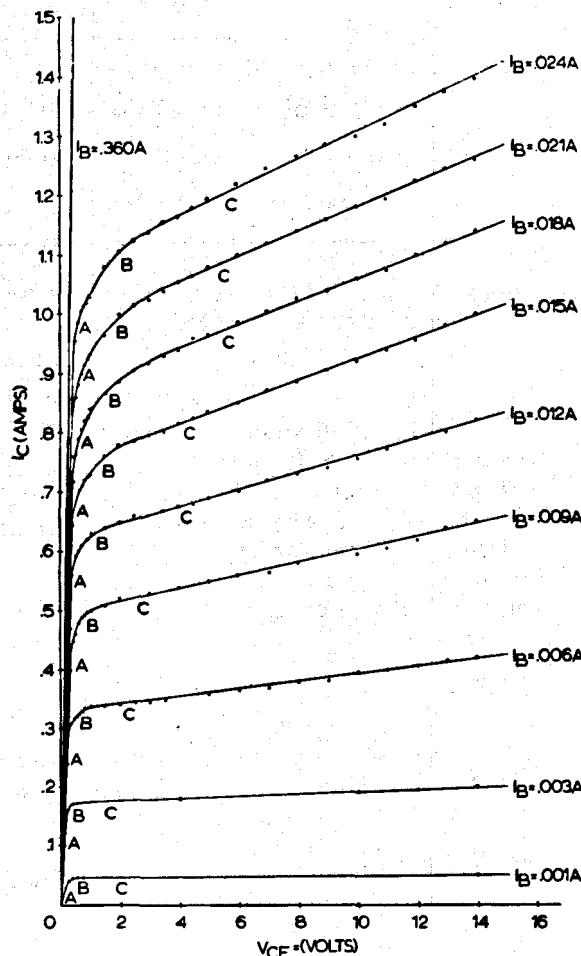


Figure 12: Nonlinear common emitter output characteristic curves for the 2N5192 transistor. The curves are plotted from the data of table 9.

The complete base-emitter model for the 2N5192 transistor wired in a common-emitter configuration is shown in figure 11. The inequalities which specify the condition of the branches when the base-to-emitter voltage is applied to the model are specified in table 8.

With the base-emitter junction of the transistor modeled, we move on to the collector-emitter junction of the transistor. The measured data which will be used in constructing the piecewise linear common emitter output characteristics is listed in table 9, which is a tabulation of collector current versus collector-to-emitter voltage drop for several base current values.

Figure 12 is the family of nonlinear curves obtained by plotting the data of table 9. Each plot of collector current (I_C) versus collector-to-emitter voltage drop (V_{CE}) for a constant base current (I_B) has two linear sections separated by a curved section (knee). These parts of each plot are labeled A, B, and C. To a good approximation, all of the A sections blend together to form the saturation characteristic curve, and the B sections may be ignored.

The C section of each plot in figure 12 is best drawn by plotting all points indicated by the data in table 9 and drawing a smooth curve through the locus for each curve. After these sketches are made, the long straight C sections may be drawn more accurately by observing the intersection of the B and C sections for each plot and deciding at which area of the locus the B section seems to level out to become the C section.

With the algorithm of figure 3, the col-

lector current intercept (value of I_C when V_{CE} equals 0) and coordinate for a collector-to-emitter voltage of 15 V may be computed for each C section. Straight lines, which intersect the saturation curve, may then be drawn through these two points for each curve to form the piecewise linear approximation to the output characteristics. Table 10 lists the slope, collector current intercept, and the value of the collector current for collector-to-emitter voltages of 15 V for each C section.

All the A sections of figure 12 may be blended into a single linear segment by obtaining a set of data points with the transistor operating in the saturation region. We guarantee that the transistor is saturated by forcing the base current to be:

$$I_B = \frac{I_C}{\beta}$$

Alternately, we can allow the base current to increase until the collector junction is forward biased by at least the threshold voltage.

For the transistor we are modeling, a base current of 360 mA caused a collector current of 3.2 A to flow with a collector-to-emitter saturation voltage of 1.0 V. Since a collector current of 3.2 A is beyond the range of our graph, the following data pairs are used in the algorithm of figure 3:

$$V_{CE} = 0.0 \text{ V} \quad I_C = 0.0 \text{ A}$$

$$V_{CE} = 1.0 \text{ V} \quad I_C = 3.2 \text{ A}$$

The slope of the linear saturation segment is calculated to be 3.2 A. A collector-to-emitter voltage drop of 0.45 V causes a collector current of 1.44 A. This voltage was chosen arbitrarily as a point which would yield a collector current coordinate within the range of the graph.

The sets of voltage and current coordinates listed in table 10 and the coordinates for the saturation segment combine to yield the piecewise linear approximation drawn in figure 13. In this figure the B sections of the curves in figure 12 have been omitted. The A sections have been blended with the saturation curve for a base current value of 360 mA. (For clarity, the sections of the graph have not been labeled.) Using this graph of the piecewise linear approximation, we may calculate the output admittance (h_{oe}), the current gain (β), and the saturation resistance equivalence for particular regions of interest.

A precise model of the collector-emitter junction for the transistor would be much more complicated than that for the base-emitter junction. Therefore, models for particular regions of transistor operation in

I_B amperes	Slope	I_C at $V_{CE} = 0$	I_C at $V_{CE} = 15$
0.001	0.000233	0.046	0.049
0.003	0.001547	0.175	0.198
0.006	0.006448	0.328	0.425
0.009	0.010380	0.496	0.652
0.012	0.014096	0.618	0.829
0.015	0.018197	0.741	1.014
0.018	0.019288	0.868	1.157
0.021	0.020451	0.976	1.283
0.024	0.021757	1.088	1.415

Table 10: The slope of each C section and sets of coordinates which are used to plot the piecewise linear approximation shown in figure 13. The collector current (I_C) is measured in amperes; the collector-to-emitter voltage is measured in volts.

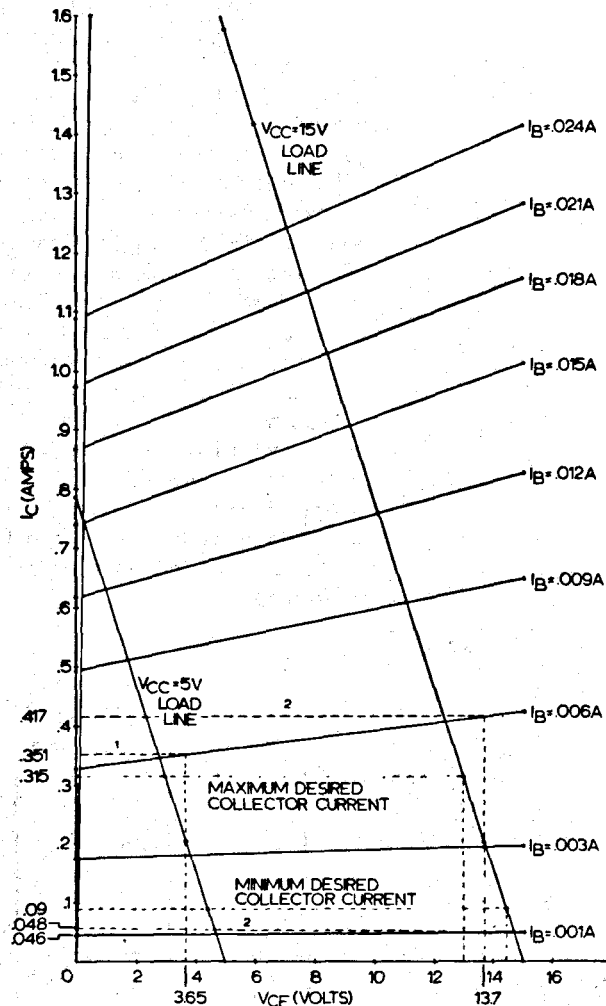


Figure 13: Piecewise linear approximation to the family of common emitter output characteristics for the 2N5192 transistor. The load lines are discussed in the text.

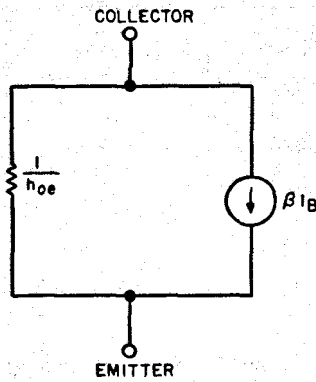


Figure 14: Simplified representation of the collector-emitter junction of a transistor wired in the common emitter configuration and operating in the active region. β is the current gain of the transistor.

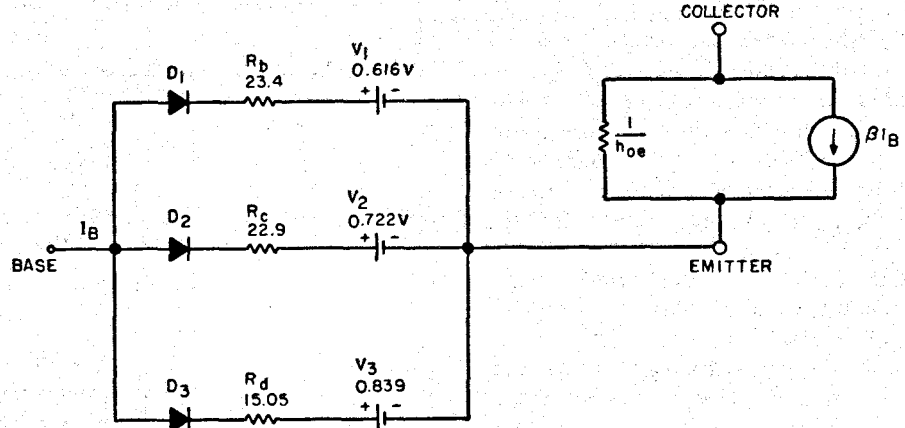


Figure 15: Complete active region model of the 2N5192 transistor. β and $1/h_{oe}$ must be determined as described in the text.

the common emitter configuration will be discussed.

Figure 14 shows a simplified representation of the collector-emitter junction of our transistor when it is wired in the common emitter configuration and operating in the active region. The collector-emitter junction is represented by the common emitter output admittance (h_{oe}) in parallel with a dependent current source whose output depends on the base current (I_B) and the DC forward current gain (β). The values for gain and admittance must be obtained from the piecewise linear approximation of figure 13 for the particular region of interest.

A complete, simplified active region model for the transistor is shown in figure 15. This model is composed of the base-emitter model of figure 11 and the collector-emitter model of figure 14. In order to illustrate the manner in which the model of figure 15 may be used to study transistor action in a practical circuit, a DC analysis of the circuit shown in figure 16 will be performed.

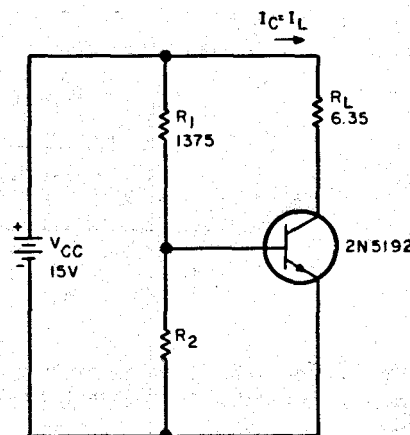


Figure 16: DC amplifier that will be analyzed as an example. A value of R_2 must be found which permits a current of 315 mA to flow through R_L .

DC Analysis

We will present a simple analysis of the circuit of figure 16 so that we may examine the preparation necessary to obtain equations that may be entered into a computer program which allows circuit simulation. Although the circuit is trivial, the techniques used to formulate the mathematical representation of the circuit are not.

Our imagined problem (example 1) is to compute a value of resistance for R_2 , by component substitution, which will allow a maximum current of 315 mA to flow through R_L . We stipulate that the current through R_L should never be less than 90 mA and that R_L will be destroyed if a current greater than 425 mA is allowed to flow through it. Of course, our simulation will not destroy anything. However, it will not be useful unless it allows us to determine the necessary value of R_2 with enough accuracy to prevent any mishaps when we actually construct the circuit.

We begin by obtaining values for inverse admittance ($1/h_{oe}$) and gain (β). Because we will experience difficulty in plotting a load line for the 15 V supply, let us compute the values of inverse admittance and gain for a 5 V supply to illustrate the technique. A load line corresponding to the voltage-versus-current characteristics of the battery load resistor combination must be superimposed on the piecewise linear collector characteristics of figure 13. The slope of the load line represents the voltage and current characteristic of resistance R_L . The equation for the load line is:

$$V_{CE} = V_{CC} - R_L I_C$$

When the collector current equals 0, the voltage drop from the collector to the emitter

equals the supply voltage or 5 V. When the voltage drop from the collector to emitter is zero, the collector current equals the source voltage divided by R_L , or 0.787 A. These two sets of coordinate pairs, ($V_{CE} = 5$ V, $I_C = 0$ A) and ($V_{CE} = 0$ V, $I_C = 0.787$ A), define the load line whose slope is the negative inverse of R_L ($-1/R_L$). The 5 V load line is drawn through these points in figure 13. Dotted lines from the minimum and maximum values of the desired collector current are drawn to the load line parallel to the voltage axis. The entire region of interest along the load line is enclosed by the base current values of 1 mA and 6 mA curves. The base current curve for 3 mA is enclosed by the region of interest.

The three base current curves associated with the region of interest will be used to calculate an average admittance value (see table 11) for the region (the slopes of the curves are shown in table 10):

An average value of the current gain is obtained by extending a dotted line in two directions parallel to the collector current axis and away from the center of the region of operation on the load line. This dotted line intersects the voltage axis and cuts through the constant base current curves which enclose the region of interest. The current gain is calculated for the load line using the collector current associated with the collector-to-emitter voltage value intersected by the dotted line and the base current curves which enclose the region of interest. See dashed line 1 and solid line 1 in figure 13. The current gain is:

$$\beta = \frac{\Delta I_C}{\Delta I_B} \bigg|_{V_{CE}} = 3.65$$

$$\beta_{ave} = \frac{0.351 - 0.046}{0.006 - 0.001} = 61$$

We now consider the load line for the 15 V supply. As before:

$$V_{CE} = V_{CC} - R_L I_C$$

When the collector current is zero, the collector-to-emitter voltage drop equals the supply voltage, which is 15 V. Setting the collector-to-emitter voltage drop to zero causes the collector current to be defined as the supply voltage divided by R_L , or 2.36 A. The point ($I_C = 2.36$ A, $V_{CE} = 0$ V) is not within the range of the coordinates of figure 13. Therefore, we must compute another point by entering the following points into the program of figure 3:

$$(V_{CE} = 15 \text{ V}, I_C = 0 \text{ A})$$

$$(V_{CE} = 0 \text{ V}, I_C = 2.36 \text{ A})$$

$$h_{oe} = \frac{\Delta I_C}{\Delta V_{CE}} \bigg|_{I_B = \text{constant}}$$

$$h_{oe(AVE)} = \frac{\text{slope } (I_B = 6 \text{ mA}) + \text{slope } (I_B = 3 \text{ mA}) + \text{slope } (I_B = 1 \text{ mA})}{3}$$

$$h_{oe(AVE)} = \frac{0.006448 + 0.001547 + 0.000233}{3}$$

$$h_{oe(AVE)} = 0.002743$$

$$\frac{1}{h_{oe(AVE)}} = 364 \Omega$$

Table 11

We may then enter a smaller value for the collector to emitter voltage into the program to obtain a collector current value within the range of the graph. For example, we would obtain a collector current value of 1.57 A for a collector-to-emitter voltage drop of 5 V.

The two points used to locate the 15 V load line are ($V_{CE} = 15$ V, $I_C = 0$ A) and ($V_{CE} = 5$ V, $I_C = 1.57$ A). This load line is also shown in figure 13. The same dotted lines for the minimum and maximum collector currents are used. The region of interest is enclosed by the same base current values as before. Therefore, the inverse admittance ($1/h_{oe}$) remains the same as it was for the 5 V load line: 364 Ω . However, the current gain has changed as signified by the dotted line extending away from the center of the region of interest on the 15 V load line and the dashed lines labeled 2. In this case:

$$\beta = \frac{\Delta I_C}{\Delta I_B} \bigg|_{V_{CE}} = 13.7$$

$$\beta_{ave} = \frac{0.417 - 0.048}{0.006 - 0.001} = 73.8 \approx 74$$

The circuit of figure 16 has been redrawn in figure 17 using the transistor model of figure 15. The current gain and inverse admittance have also been added to the simulation. The equations for this circuit are more easily written if the resistances are changed to conductances ($G = 1/R$). Therefore, the equivalent circuit has been redrawn with conductances instead of resistors in figure 18. In this figure, seven nodes have been identified as V_0 to V_6 .

The following node voltages are already known:

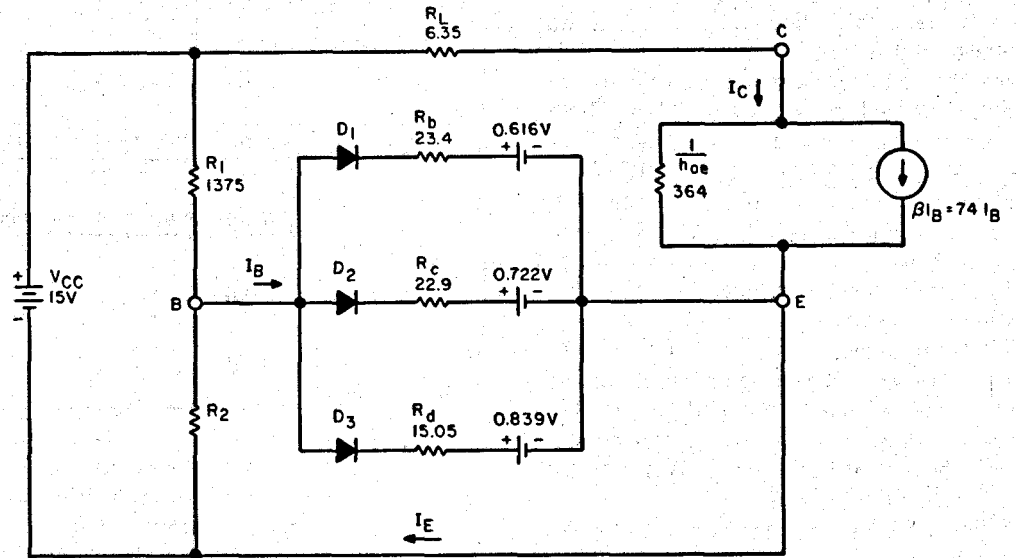
$$V_1 = V_{CC} = 15 \text{ V}$$

$$V_2 = 0.616 \text{ V}$$

$$V_3 = 0.722 \text{ V}$$

$$V_4 = 0.839 \text{ V}$$

Figure 17: Equivalent circuit for the example of figure 16.



Therefore, in order to find the load current (I_L) for a particular value of R_2 , equations must be written only for nodes 5 and 6. These two node equations are written assuming, initially, that the base-to-emitter voltage will be large enough to forward bias diodes D_1 , D_2 , and D_3 . The conditions at node 5 are given by:

$$G_1 (V_5 - V_1) + G_2 V_5 + G_b (V_5 - V_2) +$$

$$G_c (V_5 - V_3) + G_d (V_5 - V_4) = 0$$

$$\begin{aligned} & (0.00073) V_5 - (0.00073) (15) + G_2 V_5 + \\ & (0.04274) (V_5) - (0.04274) (0.616) + \\ & (0.04367) V_5 - (0.04367) (0.722) + \\ & (0.06645) V_5 - (0.06645) (0.839) = 0 \\ & (0.15359 + G_2) V_5 = 0.12456 \text{ V} \end{aligned}$$

$$V_5 = \frac{0.12456}{0.15359 + G_2} \text{ V}$$

If we let the resistance of R_2 equal 50Ω , then the conductance G_2 is equal to the inverse of that or 0.02 mhos . Therefore:

$$V_5 = \frac{0.12456}{0.15359 + 0.02} = 0.718 \text{ V}$$

The conditions at node 6 are given by

$$G_L (V_6 - V_1) + h_{oe} V_6 + \beta I_B = 0$$

To begin the analysis, we substituted the values shown in figure 18 into the node 5 equations:

With V_5 equal to the base-to-emitter voltage drop or 0.718 V , the d branch of figure 18 will be off because diode D_3 will not be

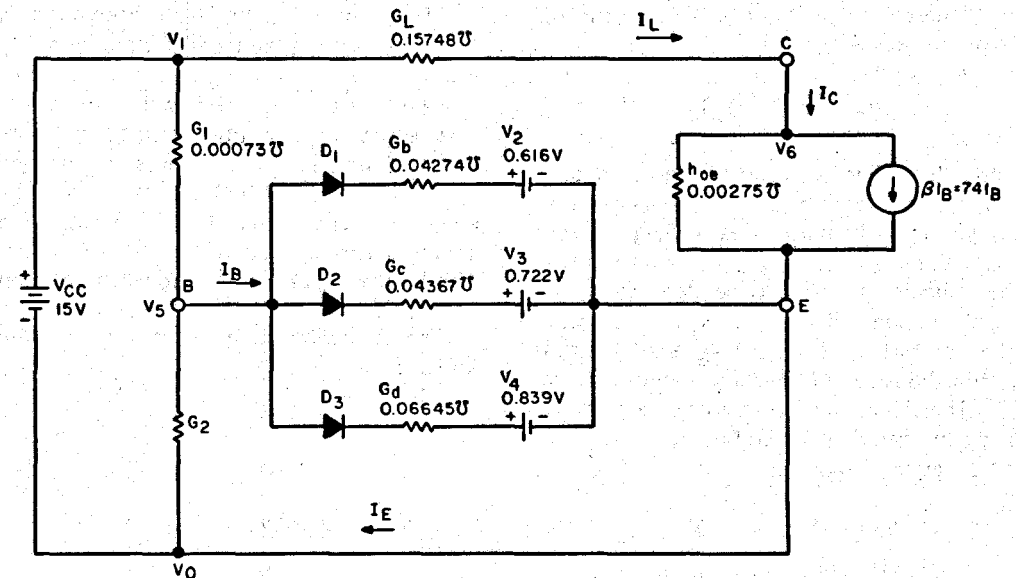


Figure 18: Equivalent circuit for the example of figure 16 using conductances instead of resistors. The seven nodes are identified as V0 through V6.

forward biased. Therefore, V_5 must be computed again with the D branch removed from the circuit and its branch conductance and voltage removed from the equation for node 5:

$$(0.08714 + G_2) V_5 = 0.06881$$

$$V_5 = \frac{0.06881}{0.08714 + 0.02} = 0.642 \text{ V}$$

With V_5 set equal to the base-to-emitter voltage drop (0.642 V), the c branch of figure 18 will also be off because diode D_2 will not be forward biased. Therefore, V_5 must be computed once again, with both the d and c branches removed from the circuit and their branch conductances and voltages removed from the equation for node 5:

$$(0.04347 + G_2) V_5 = 0.03728$$

$$V_5 = \frac{0.03728}{0.04347 + 0.02} = 0.587 \text{ V}$$

A base-to-emitter voltage of 0.587 V at node 5 will not forward bias ideal diode D_1 , the diode in branch b . Therefore, this branch is also off. This means that the base-emitter junction of the transistor is not forward biased enough to move transistor operation out of the cutoff region. Therefore, we will increase the value of R_2 in figure 17. Assume again that the base-to-emitter voltage drop will be large enough to forward bias diodes D_1 , D_2 , and D_3 , and compute the base-to-emitter voltage drop for these conditions.

Assuming a resistance of 150Ω for R_2 , G_2 equals 0.00667 mhos. The conditions at node 5 are now:

$$(0.15359 + G_2) V_5 = 0.12456$$

$$V_5 = \frac{0.12456}{0.15359 + 0.00667} = 0.777 \text{ V}$$

This is not enough to forward bias diode D_3 . Therefore, branch d is removed from the circuit:

$$V_5 = \frac{0.06881}{0.08714 + 0.00667} = 0.733 \text{ V}$$

This voltage is high enough to turn diode D_2 on. Diode D_1 is also turned on.

A base-to-emitter voltage drop of 0.733 V at node 5 will forward bias diode D_2 , the diode in branch c and diode D_1 , the diode in branch b . Therefore, these two branches of the base-emitter model will be active when the resistance of R_2 is 150Ω . The base current will be a function of the branch conductance and branch voltage in both of these branches:

$$I_B = G_b (V_5 - V_2) + G_c (V_5 - V_3)$$

$$I_B = 0.04274(0.733 - 0.616) + 0.04367(0.733 - 0.722)$$

$$I_B = 0.00548 \text{ A}$$

The current supplied by the dependent current source in the collector-emitter circuit may now be calculated:

$$\beta I_B = (74)(0.00548) = 0.40552 \text{ A}$$

The voltage at node 6 may also be calculated:

$$(0.15748) V_6 - (0.15748) (15) +$$

$$(0.00275) V_6 + 0.40552 = 0$$

$$(0.16023) V_6 = 1.95668$$

$$V_6 = 12.2 \text{ V}$$

The collector current is:

$$I_C = I_L = G_L (V_1 - V_6)$$

$$I_C = 0.15748(15 - 12.2)$$

$$I_C = 441 \text{ mA}$$

This load current is much higher than the maximum desired value of 315 mA. If we attempt to construct the circuit of figure 16 with a resistance of 150Ω for R_2 , excessive load current will flow and resistor R_L will be destroyed. We must choose a value of R_2 which is between 150Ω (excessive load current) and 50Ω (transistor does not turn on). It seems reasonable to try a value of 100Ω (since that is the value I used to set up the example), splitting the difference between these two extremes. The conditions at node 5 are:

$$V_5 = \frac{0.12456}{0.15359 + 0.01} = 0.761 \text{ V}$$

This is not enough to turn on diode D_3 . Therefore branch d is removed from the circuit:

$$V_5 = \frac{0.06881}{0.08714 + 0.01} = 0.708 \text{ V}$$

This result will not turn on diode D_2 . Therefore, branch c is removed from the circuit:

$$V_5 = \frac{0.03728}{0.04347 + 0.01} = 0.697 \text{ V}$$

This produces enough voltage to forward bias diode D_1 . Branch b of the circuit is being used. The base current under these conditions is:

$$I_B = G_b (V_5 - V_2) = 0.04274 (0.697 - 0.616)$$

$$I_B = 0.00346 \text{ A}$$

$$\beta I_B = (74)(0.00346) = 0.25618 \text{ A}$$

The following conditions are present at node 6:

$$\begin{aligned}
 (0.15748) V_6 - (0.15748) (15) + \\
 (0.00275) V_6 + 0.25618 &= 0 \\
 (0.16023) V_6 &= 2.1060 \\
 V_6 &= 13.1 \text{ V} \\
 I_C = I_L &= G_L (V_1 - V_6) \text{ mA} \\
 I_C &= 0.15748 (15 - 13.1) \text{ mA} \\
 I_C &= 299 \text{ mA}
 \end{aligned}$$

Although this load current value is closer to the desired maximum of 315 mA, we can get much closer by allowing resistor R_2 to assume larger values in smaller increments. For example, if we were to let resistor R_2 assume a value of 105 Ω , we could calculate a collector current value of 310 mA. The actual value of resistor R_2 in the circuit set up for this example was 100 Ω . The model suggests that we would obtain a load current of 299 mA for this value of R_2 . Having assumed a value of 105 ohms for R_2 and having obtained a load current of 310 mA for this value, we would not attempt to find the precise value of R_2 necessary to insure a model load current of 315 mA. The resolution of our model is not good enough to warrant such precise calculations. Instead we should probably settle for a current of 310 mA for a resistance of 105 ohms.

Experience with the model over a period of time would allow us to determine if it were accurate enough for our needs. If it were a model to be used in a one time simulation and experiment, we would simply insert a 100 Ω potentiometer into the circuit as R_2 and increase the resistance from 50 to 100 Ω while monitoring the collector current. If, however, the model were found to be generally useful, the model parameters could be stored and recalled as needed.

This example has been presented in order to illustrate the type of data which may be obtained from the piecewise linear characteristic curves and the technique of repetitiously solving a set of equations to simulate component variations in a circuit which is being analyzed. For the example given, two simple node equations were written for an equivalent circuit which included the model of an active device.

In general, however, we may expect that the circuits we wish to simulate will force us to consider many equations simultaneously. When the analysis we wish to perform is based on a linear model, the unknown quantities will usually appear only to the first power, and the coefficients in the equations relating n unknowns can be expressed in the form:

$$\text{equation 1 } a_1 X_1 + a_2 X_2 + \dots + a_n X_n = Y_1$$

$$\text{equation 2 } b_1 X_1 + b_2 X_2 + \dots + b_n X_n = Y_2$$

$$\begin{aligned}
 &\vdots \\
 &\vdots \\
 &\vdots
 \end{aligned}$$

$$\text{equation } n \quad m_1 X_1 + m_2 X_2 + \dots + m_n X_n = Y_n$$

Each quantity X is an unknown quantity, the m terms are the coefficients of the unknowns, and the Y terms are the right hand sides of the equations. Methods of solution of simultaneous equations can be found in many high school and college math texts.

Complete NPN Transistor Model

We have successfully constructed an active region model for an NPN transistor wired in a common emitter configuration. At this time we shall include in our model the other regions of operation. The collector-emitter model for transistor operation in the saturation region is shown in figure 19. The value of the saturation resistance is:

$$R_{\text{sat}} = \frac{V_{\text{CE,sat}}}{I_{\text{C,sat}}} = \frac{1.0 \text{ V}}{3.2 \text{ A}} = 0.31250 \Omega$$

The variables $V_{\text{CE,sat}}$ and $I_{\text{C,sat}}$ are the measured values for the collector-to-emitter voltage drop and the collector current when the base current is equal to or greater than the collector current divided by the current gain (I_C/β). The values for $V_{\text{CE,sat}}$ and $I_{\text{C,sat}}$ shown above were discussed when the piecewise linear approximation to the output characteristics was presented earlier. The saturation region representation shown in figure 19 may replace the active region representation in the model of figure 15 when it is necessary to simulate a transistor operating only in the saturation region.

No partial model is needed to represent the cutoff region. We simply insure that no current flows in the base and collector branches to simulate this condition. We have now accumulated enough information to construct a three region transistor model as shown in figure 20. In this figure, the base-emitter junction is represented by a single branch in order to simplify the discussion that follows.

Several of the components in figure 20 must take on one of two possible values depending on the region of operation. The possible values which these components may assume are shown in parentheses next to the components. It is necessary to specify alternate values in a multiple region model so that appropriate parts of the circuit will be-

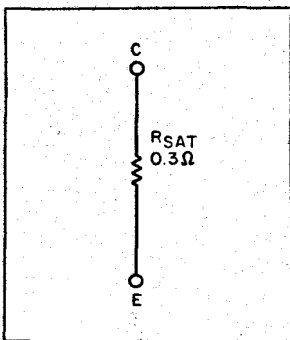


Figure 19: Saturation region model for the piecewise linear approximation of figure 13.

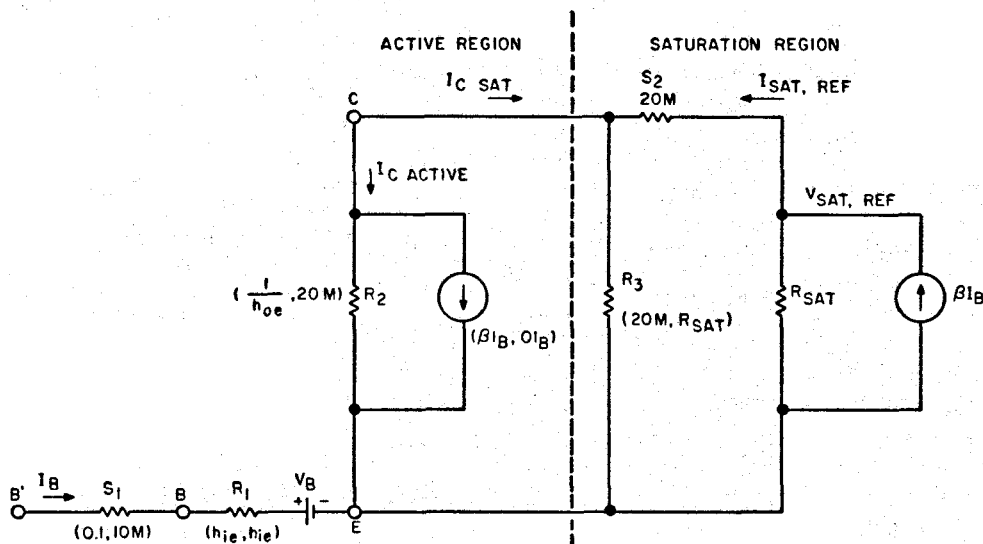


Figure 20: Three region NPN transistor model.

come enabled or disabled as transistor operation moves from one region to another.

Two of the resistors in figure 20 act as switches to control the current flow through the model. These resistors are labeled S_1 and S_2 . The operation of the three region model depends on the states of S_1 and S_2 . When the transistor is cut off, both S_1 and S_2 are off. When the model is operating in the active region, resistor S_1 is on but resistor S_2 is off. When the transistor is in the saturation region, both S_1 and S_2 are on.

The action of the switches is not straightforward. Switch S_1 controls the current flow through itself and resistors R_1 and R_2 . Switch S_2 controls the current flow through resistors R_1 , R_2 , and R_3 , but does not control the current through itself. The switches control the current through various components by selecting one of the two values specified for the components under their control. The state of each switch depends on the direction of the flow of current through the switch.

If, during an analysis, the base current tends to flow in a direction which is opposite to the arrow labeled I_B , switch S_1 will assume a value of $10\text{ M}\Omega$ (off state). This will cause resistor R_1 to assume a value of h_{ie} (R_1 is equal to the common emitter input impedance for all possible switch states). Resistor R_2 assumes a value of $20\text{ M}\Omega$ and the dependent source in the section of the model labeled *active region* assumes a value of βI_B .

With the base-emitter junction cut off, the collector-to-emitter voltage drop will be greater than or equal to $V_{sat, REF}$. Therefore, current will flow through switch S_2 in a direction opposite to the arrow shown as $I_{sat, REF}$. This will cause switch S_2 to assume the off state and resistor R_3 to

be $20\text{ M}\Omega$. Under the above conditions, the transistor is cut off.

When the base-emitter junction is forward biased, base current will flow in the direction indicated by I_B . Switch S_1 will then assume a value of 0.1Ω (on state), resistor R_1 will be the defined input impedance, R_2 will be the inverse of the output admittance ($1/h_{oe}$), and the dependent source in the section of the model labeled *active region* will assume a value of βI_B . As long as the collector to emitter voltage drop remains greater than or equal to $V_{sat, REF}$, switch S_2 will remain off, and the model will behave as if the section of the collector-emitter model labeled *saturation model* were not present.

Any time $V_{sat, REF}$ becomes greater than the collector-to-emitter voltage drop, switch S_2 will assume the on state, forcing resistor R_2 to assume a value of $20\text{ M}\Omega$, resistor R_3 to assume a value equal to the saturation resistance (R_{sat}), and the current gain of the dependent source in the *active region* section of the model to be zero. (A β of zero will result in a current of 0 A .) Under these conditions, the transistor is operating in the saturation region and the model behaves as if the section of the model labeled *active region* is not present.

The three region model of figure 20 need be used only when necessary. Many analyses may be performed with a simplified equivalent containing only an active region or saturation region representation of the collector-emitter circuit.

This concludes our discussion of bipolar transistors. We will now take a cursory look at modeling two other types of nonlinear devices: the field effect transistor, and the silicon controlled rectifier. These discussions will be very general and expect the reader to have a working knowledge of the devices.

Field Effect Transistor Models

The three variables which must be represented in a graph of the family of output characteristic curves for a field effect transistor (FET) are the drain current (I_D), the voltage drop from the drain to the source (V_{DS}), and the voltage drop between the gate and the source (V_{GS}). Figure 21 shows a piecewise linear approximation to the output characteristics of a field effect transistor. The equation for a load line depends on the supply voltage at the drain, the load resistance, and the drain current:

$$V_{DS} = V_{DD} - I_D R_L$$

The intercepts of this equation are:

$$\begin{aligned} V_{DS} &= V_{DD}, I_D = 0 \\ I_D &= \frac{V_{DD}}{R_L}, V_{DS} = 0 \end{aligned}$$

We will use the piecewise linear output characteristic for the field effect transistor to calculate the parameters in table 12. These parameters will provide the information we need to construct a model for the field effect transistor.

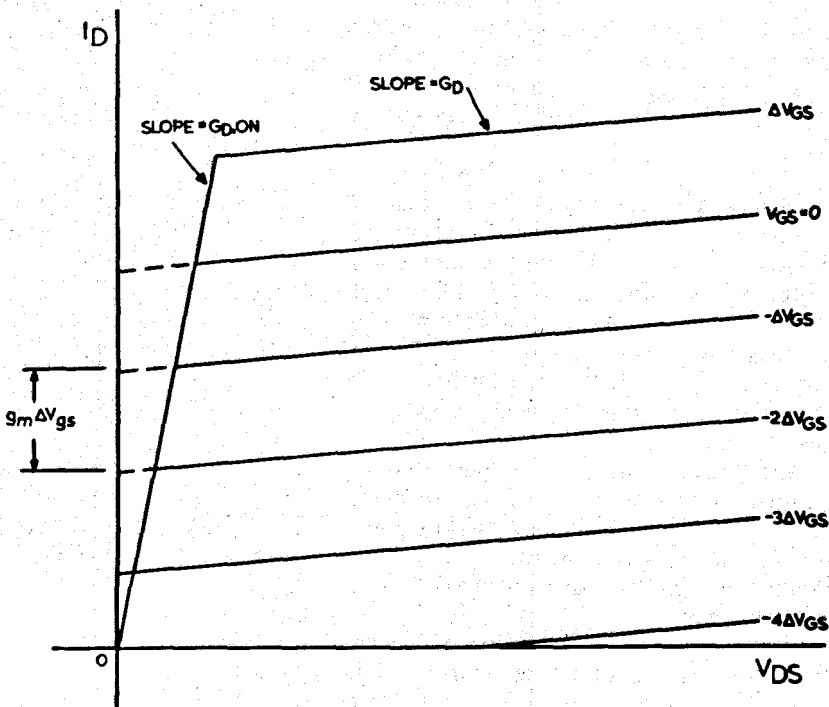


Figure 21: Piecewise linear approximation to the output characteristic of a field effect transistor (FET). This is a general family of curves which must be labeled to simulate the characteristics of the desired transistor.

$R_D = \frac{\Delta V_{DS}}{\Delta I_D} \Big _{V_{GS} = \text{constant}}$	with dimensions ohms
$g_m = \frac{\Delta I_D}{\Delta V_{GS}} \Big _{V_{DS} = \text{constant}}$	with dimensions mho
$R_{D,on} = \frac{V_{DS}}{I_D}$	at the origin
R_D	drain resistance
G_D	drain conductance
g_m	transconductance
$R_{D,on}$	on drain resistance
$G_{D,on}$	on drain conductance

Table 12: Parameters to be calculated for the linear model of the field effect transistor.

Figure 22 is one of several models we may choose to represent a junction field effect transistor (JFET). Consider diode D_1 . This diode allows the breakpoint at $(-4\Delta V_{GS}, V_{DS})$ to exist in the model. Diode D_1 prevents the drain current from becoming negative when the drain to source voltage is reversed. As long as the drain current is positive, diode D_1 acts as a short circuit and normal operation occurs. If, however, the drain current becomes negative, diode D_1 will act as an open circuit.

When the gate to source voltage drop is 0 V and the drain to source voltage drop is positive, but small enough so that the current through conductance $(G_{D,on} - G_D)$ is less than the current from the dependent source S_2 , then diode D_2 acts as a short circuit. The output circuit then consists of the conductances G_D and $(G_{D,on} - G_D)$ in parallel. The model behaves as if the dependent current source S_2 is not in the circuit. Under these conditions, the drain conductance is:

$$G_D + (G_{D,on} - G_D)$$

which is equivalent to:

$$G_{D,on}$$

The drain resistance for the on state ($R_{D,on}$) is an important parameter in switching applications where the transistor is driven heavily in the on state.

After the drain-to-source voltage drop (V_{DS}) is increased so that the current through the conductance $(G_{D,on} - G_D)$ becomes equal to the current through dependent source S_2 , any further increase in the drain current

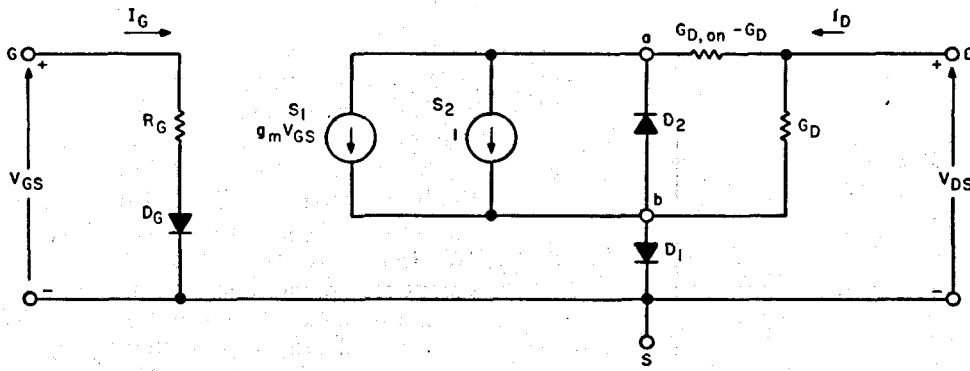


Figure 22: A general junction field effect transistor model.

must be through conductance G_D . This is because the current through the conductance ($G_{D,on} - G_D$) can never be greater than the current through dependent source S_2 . This is a result of the slope G_D in the output characteristic piecewise linear approximation. In figure 21 the intersection of the gate-to-source voltage drop with a value of zero plot with the drain current axis occurs at the current value of source S_2 . When the gate-to-source voltage drop is not equal to zero, dependent current source S_2 combines with dependent current source S_1 to provide a current equal to $(1 + g_m V_{GS})$.

If the gate-to-source voltage drop is negative, the gate channel diode (D_G) acts as an open circuit. When the gate-to-source voltage drop is positive, diode D_G represents a short circuit and resistor R_G is a model of the resistance of the forward biased gate channel diode of the field effect transistor. If the resistance R_G and the diode D_G are removed from the junction field effect transistor (JFET) model in figure 22, the model may be used to represent a metal oxide silicon field effect transistor (MOSFET). The gate of a MOSFET is insulated from the channel. Therefore, the model must indicate this with an open circuit between the gate and all other elements of the model.

The discussion of the field effect transistor model has been much briefer than that of the bipolar transistor. Furthermore, specific values for the model elements have not been given. The reason for this is that we have presented many techniques during the bipolar transistor discussion. In general, many different models are required to construct a practical electronic circuit. General examples which may be stored in computer memory, recalled, and altered as required for circuit simulations are more valuable. We will present one more general model to whet your appetite.

Silicon Controlled Rectifier Model

The current versus voltage characteristic of a typical silicon controlled rectifier (SCR) is shown in figure 23. This figure shows that the forward breakover voltage (V_{BO}) is a function of the cathode-gate current. Currents introduced into the gate terminal may be used to control the anode to cathode breakover voltage. Once the silicon controlled rectifier is turned on by a gate triggering cur-

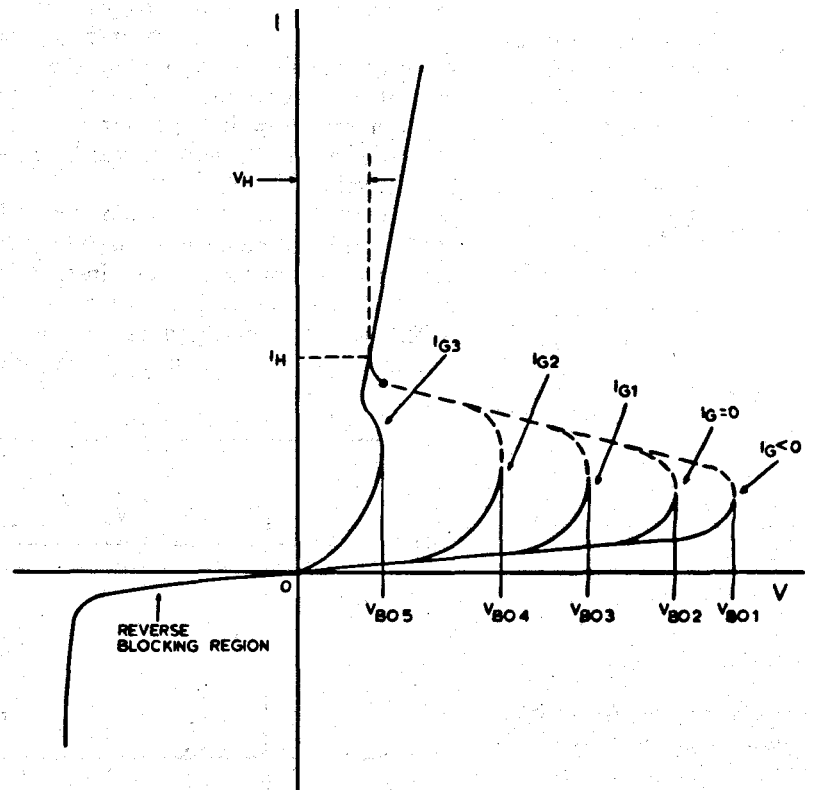


Figure 23: Current versus voltage characteristics of a silicon controlled rectifier. The breakover voltage (V_{BO}) is a function of the gate current (I_G).

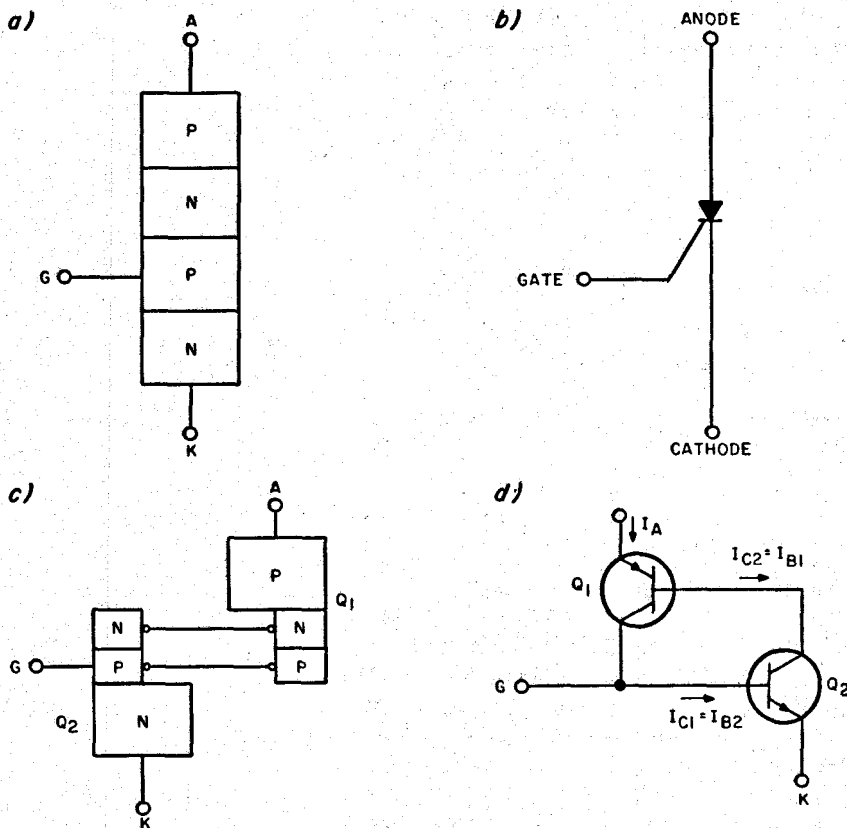


Figure 24: Silicon controlled rectifier. Figure 24a is the basic construction of the rectifier; 24b is the schematic representation; 24c is the equivalent construction; and 24d is the equivalent schematic symbol.

rent or voltage adequate to lower the break-over voltage to less than the voltage applied between the anode and cathode, it latches into the *on* state. Further trigger signals at the gate terminal, positive or negative, have no affect on the anode current. The anode current will continue to flow until it is reduced below the holding current (I_H) or until the anode voltage is reduced below the holding voltage (V_H).

The model for the silicon controlled rectifier will not be taken from the current-versus-voltage characteristics. Instead, we note that any four layer device, such as the SCR, can be represented as an interconnected NPN and PNP transistor if we consider

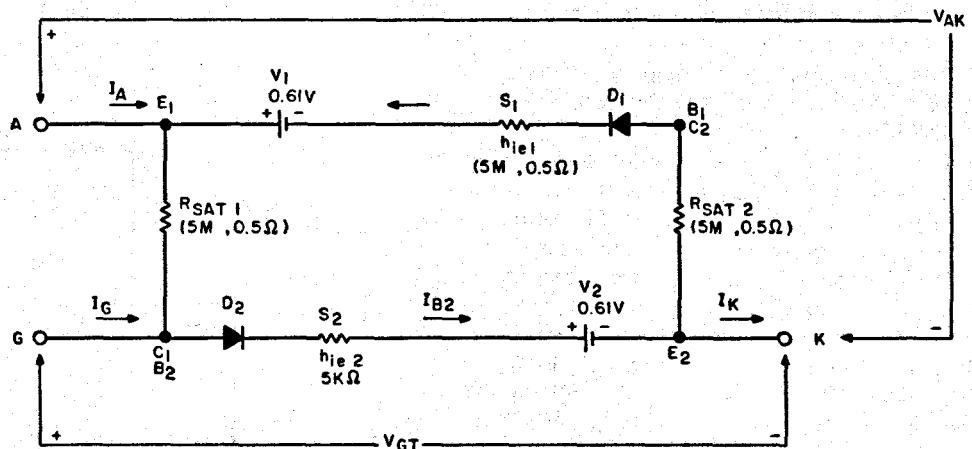
the two center regions separated as in figure 24c.

We may approximate a silicon controlled rectifier with the two transistor model of figure 24d. These transistors are connected so that the collector of transistor Q_1 drives the base of transistor Q_2 and the collector of transistor Q_2 drives the base of transistor Q_1 . The regenerative feedback in the transistor circuit simulates internal thyristor feedback. Our model of the silicon controlled rectifier will reflect these factors and will consist essentially of two simple transistor models with the active regions omitted.

The model we will use is shown in figure 25. The parameters given for the elements in the model are suggested values with which an analysis may be initiated. These values should be altered as necessary to allow the model to more closely simulate a particular rectifier. As with the transistor model discussed earlier, there are two switches in the model which control the activity of the circuit. Switch S_1 controls only the input impedance (h_{ie1}). Switch S_2 controls the resistance values at saturation (R_{sat1}, R_{sat2}).

When the anode voltage is high enough to permit holding current, switch S_1 will change the input impedance (h_{ie1}) from a value of $5\text{ M}\Omega$ (*off* state) to a value of 0.5Ω (*on* state). As long as the gate current is less than the minimum gate trigger current required to cause the silicon controlled rectifier to switch from the nonconducting to the conducting state for the applied anode voltage (I_{GT}), the rectifier will remain cut off. As soon as the gate voltage is greater than voltage source V_2 , the trigger current will flow. Switch S_2 will then force both R_{sat1} and R_{sat2} to change from $5\text{ M}\Omega$ (*off* state) to 0.5Ω (*on* state). When this occurs, full anode current flows from node E_1 through node B_1/C_2 to node E_2 . In addition, the current flowing through resistor R_{sat1} continues to supply gate current to node C_1/B_2 as long as the anode

Figure 25: Model of the silicon controlled rectifier. The equivalent circuit model represents two simple transistor models with the active regions omitted.



voltage is sufficiently more positive than the cathode voltage.

Conclusion

This introduction to computer aided design has been presented with emphasis on modeling techniques. The general models discussed reinforce the concept of storing models in memory. The stored models may be recalled as they are needed in a simulation. As the models are recalled, their parameters may be adjusted so that the models represent active devices to be connected in hard-wired circuits.

Then, the systems of equations which are generated by the circuit being analyzed may be processed with a computer or calculator program designed to produce solutions in a format that yields the information required by the designer.

The designer is allowed to attempt destructive voltage and current levels without destroying valuable components. In addition, the designer may construct software circuits and predict hard-wired results with-

out having the circuit components on hand. An electronic experimenter may invest capital in a small computer and design with software instead of purchasing many individual components which must be stored, and which could be destroyed or lost.■

BIBLIOGRAPHY

Matrix Algebra for Engineers, Gere, J M and Weaver, W Jr, published by Van Nostrand Reinhold Co, 450 W 33rd St, NY NY 10001.

Engineering Circuit Analysis, Hayt, W H Jr and Kemmerly, J E, published by McGraw-Hill Inc, POB 582 Hightstown NJ 08520.

Electronic Circuits: Physical Principles, Analysis and Design, by Chirlian, Paul M, published by McGraw-Hill Inc. POB 582, Hightstown NJ 08520.

Measuring Methods and Devices In Electronics, Beerens, A C J, published by Hayden Book Co. Inc., NY NY.

Network Computer Analysis, Zobrist, G W, published by Boston Technical Publishers, Cambridge MA.

About the Authors

Richard Rosenbaum (61-04 Little Neck Parkway, Little Neck NY 11362) is a full time computer consultant involved with systems ranging from micro-processors to large scale computers. He is currently working with computer systems in the banking industry. On the personal side, Rich is interested in computer-assisted musical composition and performance.

Michael Wimble (6026 Underwood Avenue, Grand Rapids IA 52404) is presently a private consultant in all aspects of computing, but particularly in industrial uses. His company, Wimble Robotics, is in the process of designing specialized robots for automotive manufacturing, quality control inspection, and process control. For the previous eight years he was a consultant in the area of business applications of computers. His personal system is all homebrew and is based mainly on the Texas Instruments 9900 micro-processor.

Harry Tennant (1001 W. Oregon TI, Urbana IL 68801) is currently working on his PhD in Computer Science to be completed in 1979, developing a natural language question-answer system called JETS. He is also studying techniques for evaluating natural language systems so that their performance can be more accurately described. He received his Master's degree from the University of Illinois at Chicago Circle in 1977, designing and building a natural language system for his thesis. Currently he is also engaged in writing a book on natural language processing.

Stephen P Smith (POB 841, Parksley VA 23421) leads the Computer Sciences Corporation support team attached to the range safety office at NASA Wallops Flight Center, where he and his team of analysts develop analytical methods and construct digital simulations of flight paths, flow fields and structural responses of rockets and aircraft. Stephen's pet project as an amateur is a PASCAL compiler for a personal computer.

Fred R. Ruckdeschel (773 John Glen Blvd, Webster NY 14580) is a Principal Scientist at the Xerox Corporation in Webster, New York, where he has been employed for the past eleven years. During this time he has been involved in physics and management. His first experience in computing was with the IBM 650. He later performed extensive thermal simulations using the IBM 360. More recently he has turned his interests to microcomputers for both data acquisition and personal use. Dr. Ruckdeschel's hobbies include sailing, wood working, digital electronics, programming, and haunting computer stores.

About the Authors, continued

John Webster (University of New Brunswick, Fredericton NB, CANADA E3B 5A3) received a degree in 1967 in Mass Communications from the University of Miami, Coral Gables, Florida. He then went to work for the Ontario Institute for Studies in Education in Toronto where he designed and administered a Media Services Centre. In 1971 he moved to the University of New Brunswick in Fredericton, New Brunswick to set up a department of Audio Visual Services. He is presently the Director of Audio Visual Services at the University of New Brunswick where he and his staff labor diligently to assist faculty, staff, and students in applying technology to educational communications situations.

Len Anderson (10048 Lamark St, Sun Valley CA 91352) has 26 years of electronics experience, 18 as a system and circuit design engineer. A member of IEEE CAS and Computer Society, he is a strong advocate of computer aided design and contributed several FORTRAN design programs to RCA Central Engineering. Formerly Senior Staff Engineer at Teledyne Electronics, his last job involved a 6800 based military test set. Now a writer and consultant, he has had over twelve articles published on linear and digital circuit design. A number crunching microcomputer system is in the works to handle more computer aided design tasks.

Robert C. Arp, Jr. (3961 Acapulco Drive, Campbell CA 95008) received a BSEE degree from California State University, Sacramento in June 1974. While enrolled in the Electrical Engineering graduate program the following year, he was editor of the CSUS engineering journal. After ten years of study in bioengineering and languages at three universities, he left the CSUS in June 1975 to accept a position as PMOS Product Engineer at National Semiconductor Corporation in Santa Clara, California. In May 1977 he accepted a position as Manufacturing Engineer of the Product Insertion Group at Bently Nevada Corporation of Minden, Nevada. Robert has written for several computer, electronics, and ham radio magazines. His primary interests, at the present time, are applications, microprocessors, microcomputers, and computer-aided design.

BYTE Books

Edmond Kelly, Jr., publisher

Blaise W. Liffick, technical editor

Raymond G. A. Cote, assistant technical editor

Richard Shuford, assistant technical editor

William H. Hurlin, production editor

Patricia Curran, production editor

Risa Swanson, production artist

E. S. Associates, production art

Techart Associates, drafting

Walter Banks, University of Waterloo, CCNC, bar codes

George Banta Company, printing

Dawson Advertising Agency, cover art

Simulation

Programming Techniques is a collection of the best articles from BYTE magazine plus new material concerned with the art and science of computer programming. The basic principle of the series is to provide the personal computer user with sufficient background information to write and maintain programs effectively.

SIMULATION, the second book in the series, contains articles dealing with aspects of specific types of simulation. Both theoretical and practical applications are included. Particularly stressed is simulation of motion, including wave motion and flying objects. The realm of artificial intelligence is explored, along with simulating robot motion with the micro-computer. One article deals with simulation of electronic circuits on the computer.

Other books in the Programming Techniques series are:

Program Design ISBN 0-931718-12-0

Numbers in Theory and Practice ISBN 0-931718-14-7

Bits and Pieces ISBN 0-931718-15-5

